Pós-Graduação em Ciência da Computação

Specifying Design Rules in Aspect-Oriented Systems

Por

Alberto Costa Neto

Tese de Doutorado

Universidade Federal de Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE - PE

05 de Março de 2010

UNIVERSIDADE FEDERAL DE PERNAMBUCO

CENTRO DE INFORMÁTICA

PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ALBERTO COSTA NETO

Specifying Design Rules in Aspect-Oriented Systems

ESTE TRABALHO FOI APRESENTADO À PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DO CENTRO DE INFORMÁTICA DA UNIVERSIDADE FEDERAL DE PERNAMBUCO COMO REQUISITO PARCIAL PARA OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIA DA COMPUTAÇÃO.

ORIENTADOR: *Paulo Henrique Monteiro Borba*
CO-ORIENTADOR: *Fernando José Castor de Lima Filho*

RECIFE – PE

05 de Março de 2010

Costa Neto, Alberto

Specifying design rules in aspect-oriented systems / Alberto Costa Neto. - Recife: O Autor, 2010.

xii, 122 folhas : il., fig., tab.

Tese (doutorado) – Universidade Federal de Pernambuco. CIn. Ciência da Computação, 2010.

Inclui bibliografia e apêndice.

1. Engenharia de software. 2. Desenvolvimento orientado a aspectos. 3. Projeto de software. 4. Linguagem de programação. I. Título.

005.1 CDD (22. ed.) MEI2010 – 063

Tese de Doutorado apresentada por **Alberto Costa Neto,** a Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **"Specifying Design Rules in Aspect-Oriented Systems"**, elaborada sob a orientação do **Prof. Paulo Henrique Monteiro Borba** e aprovada pela Banca Examinadora formada pelos professores:

Prof. Marcelo Bezerra d'Amorim
Centro de Informática / UFPE

Prof. Juliano Manabu Iyoda
Centro de Informática / UFPE

Prof. Sergio Castelo Branco Soares
Centro de Informática / UFPE

Prof. Marco Túlio de Oliveira Valente
Departamento de Ciência da Computação / UFMG

Prof. Uirá Kulesza
Departamento de Informática e Matemática Aplicada /UFRN

Visto e permitida a impressão.
Recife, 05 de março de 2010.

Prof. NELSON SOUTO ROSA
Coordenador da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco.

Dedico este trabalho à minha família.

Agradecimentos

Depois de muitas viagens de Aracaju para Recife e de muito trabalho, consegui alcançar o objetivo de concluir o doutorado. Algumas pessoas participaram desta fase da minha vida, às quais agradeço abaixo:

- a Deus pela presença na minha vida;
- à minha esposa Allana que foi fundamental por compreender a importância deste trabalho para mim, abdicando dos nossos momentos juntos e proporcionando as condições necessárias para que eu pudesse terminar este trabalho;
- a meu filho Albertinho, que desde pequeno foi um grande guerreiro e foi uma grande fonte de motivação para conclusão do doutorado;
- a meus pais por toda a dedicação e por não medirem esforços para que eu tivesse uma boa educação;
- a minhas irmãs e sobrinha Natália por todos os momentos de alegria;
- ao professor Paulo Borba pela paciência e preocupação com o meu trabalho, e por todos os ensinamentos que me passou durante o doutorado;
- ao professor Fernando Castor pelo incentivo, e principalmente pela troca de idéias na fase final do doutorado;
- ao grande amigo Rohit pelo apoio e motivação nos momentos mais difíceis do doutorado e pela companhia nos momentos de distração;
- aos amigos Sergio, Tiago, Dósea, Márcio, Rodrigo e Vander pela ajuda e trabalhos conjuntos que contribuíram para esta tese;
- aos professores do DCOMP/UFS pelo incentivo a cursar o doutorado;
- aos demais membros do Software Productivity Group e amigos que me ajudaram a chegar até aqui.

Resumo

Programação Orientada a Aspectos é conhecida como uma técnica para modularização de interesses transversais. Entretanto, construções que visam apoiar a modularidade transversal podem quebrar a modularidade de classe. Como consequência, os desenvolvedores de classes enfrentam problemas de modificabilidade, desenvolvimento em paralelo e entendimento, porque precisam estar conscientes da implementação dos aspectos sempre que forem desenvolver ou dar manutenção em uma classe. Ao mesmo tempo, aspectos são vulneráveis a mudanças nas classes, já que não existe um contrato especificando os pontos de interação entre estes elementos. Estes problemas podem ser mitigados através de Regras de Projeto entre classes e aspectos. Nós apresentamos uma linguagem para especificação de Regras de Projeto (LSD) e exploramos seus benefícios desde as fases iniciais do processo de desenvolvimento, especialmente com o objetivo de dar apoio ao desenvolvimento modular de classes e aspectos. Nós discutimos como nossa linguagem melhora a modularidade transversal sem quebrar a modularidade de classe. Além disso, especificamos a semântica da linguagem em Alloy. A linguagem é implementada através de uma extensão do abc (AspectBench Compiler), tornando mais fácil expressar e checar muitas das Regras de Projeto encontradas em sistemas Orientados a Aspectos. Nós avaliamos LSD usando o sistema Health Watcher como estudo de caso e comparamos com abordagens existentes.

Palavras-chave: Programação Orientada a Aspectos, Regras de Projeto, Modularidade.

Abstract

Aspect-Oriented Programming is known as a technique for modularizing crosscutting concerns. However, constructs aimed to support crosscutting modularity might actually break class modularity. As a consequence, class developers face changeability, parallel development and comprehensibility problems, because they must be aware of aspects whenever they develop or maintain a class. At the same time, aspects are vulnerable to changes in classes, since there is no contract specifying the points of interaction amongst these elements. These problems can be mitigated by using adequate Design Rules between classes and aspects. We present a Design Rule specification language (LSD) and explore its benefits since the initial phases of the development process, specially with the aim of supporting modular development of classes and aspects. We discuss how our language improves crosscutting modularity without breaking class modularity. Besides, we specify the language semantics in Alloy. The language is implemented through an extension of abc (AspectBench Compiler), making it easy to express and check most of the Design Rules found in Aspect-Oriented systems. We evaluate the language using the Health Watcher system as case study and compare it with existent approaches.

Keyworks: Aspect-Oriented Programming, Design Rules, Modularity.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Modularity is an important quality attribute that guides software development. A software system can be more or less modular depending on the design decisions made by developers. According to David Parnas [78], the benefits expected of modular programming are:

1. Managerial-development time should be shortened because separate groups would work on each module with little need for communication;

2. Product flexibility – it should be possible to make drastic changes to one module without a need to change others;

3. Comprehensibility – it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood.

The search for more modular software involves design decisions that are influenced by the available forms of expressing a design. There are several paradigms, *e.g.*, Imperative Programming, Functional Programming and Object-Oriented Programming (OOP), that define concepts and elements that support the modeling of software through some specific theory. Each one modularizes concerns using different strategies.

Aspect-Oriented Programming (AOP) [52] is a new paradigm that extends OOP with the concept of aspects focused on implementing crosscutting concerns, that can lead to a more modular design. The implementation of these crosscutting concerns without aspects is scattered throughout the modules of a system, tangled with code implementing other concerns. AOP is a technique for modularizing crosscutting concerns [52]. Among others, Logging, distribution, tracing, security, persistence, and transactional management are accepted as examples of crosscutting concerns well addressed by AOP [89, 90].

1.1 Problem

AOP leads to modularity problems such as a high coupling between classes and aspects. In this context, several authors argue that, in aspect-oriented systems, in order to reason about the system modules (e.g., classes, aspects), it is necessary to consider all aspect

implementations [94, 22, 91]. In the presence of aspects, class modularity is compromised because, when evolving a class (for instance, extracting a method), it might be necessary to analyze the implementation of existing aspects, instead of analyzing only the class and the interface of other referred classes. Aspects might refer to class implementations (like intercepting a call to a method `m2` within a method `m1`, as illustrated in Listing 1.1).

Listing 1.1: Aspect dependent on a method call.

```
1  public class C {
2     public void m1() {
3        m2();
4     }
5     public void m2() {...}
6  }
7
8  public aspect A {
9     pointcut callToM2() : call(* C.m2()) && withincode(* C.m1());
10    after() : callToM2(){...}
11 }
```

In fact, by referring to implementation class details in aspects, one can inhibit modular reasoning and compromise changeability, requiring class modifications to be fully aware of the aspects affecting the class. For example, the developer might need to check all aspects in order to confirm that the method extraction will not lead to missing joinpoints. Therefore, constructs aimed to support *crosscutting modularity* might actually break *class modularity* [84], creating mutual dependencies between classes and aspects, as shown by Figure 1.1.

This situation is exemplified in Listing 1.2, where the call to method `m2` was extracted to the body of method `m3`. Even though this refactoring preserves the behavior of class `C` (from an Object-Oriented perspective), in the presence of aspect `A`, the overall behavior of the application will be affected. So, the change to class `C` requires an adaptation to pointcut `callToM2` from aspect `A`. Although this is a simple example, we can notice that it is not possible to reason about class `C` in isolation, differently from a purely OO program.

Listing 1.2: Breaking an aspect with a method extraction.

```
1  public class C {
2     public void m1() {
3        m3();
4     }
5     public void m3() {
6        m2();
7     }
8     public void m2() {...}
9  }
10
11 public aspect A {
12    pointcut callToM2() : call(* C.m2()) && withincode(* C.m1());
13    after() : callToM2(){...}
14 }
```

The limitations of AOP (breaking class modularity) can be mitigated by using adequate Design Rules specifying the interface between classes and aspects, as discussed in other works [94, 65]. Design rules are necessary to reduce such new dependencies in Aspect-Oriented (AO) systems (illustrated in Figure 1.2). They are not just guidelines and recommendations: they generalize the notion of information hiding interfaces and

Figure 1.1: Dependencies between classes and aspects

must be rigorously obeyed. Besides being useful for verification purposes, they serve as a guideline for developers since the initial phases of the development process.

Figure 1.2: Dependencies between classes, aspects and design rules

Sullivan *et al.* [94] show how an AO program can benefit (in terms of modularity) from the adoption of design rules since the early stages of development, when compared to an approach in which class and aspect developers do not define any rule before implementation (*oblivious approach*). They noticed that a system developed based on pre-established design rules was less complex and presented less dependencies between classes and aspects. One problem of the design rule approach described by Sullivan *et al.*. is that design rules are specified informally, in natural language. This may lead to sometimes verbose, incomplete, inconsistent, and ambiguous specifications. Also, natural language cannot be checked automatically due to its expressiveness. In a subsequent work [43], Griswold *et al.* introduce the idea of expressing design rules using AspectJ through the definition of Crosscutting Programming Interfaces (XPI). Although it is possible to check part of the design rules using XPIs, the use of a language not designed to this purpose leads frequently to complex specifications because the contracts are mechanically checked using aspects. However, to the best of our knowledge, none of the previous approaches [94, 43, 65] proposes a language with the specific purpose of describing design rules in AO systems.

1.2 Solution

We present a Language for Specifying Design Rules (LSD) [72] that improves the modularity of AO systems implemented with AspectJ [51]. This is achieved by specifying, as an interface, the essential structure and behavior that each developed component must provide with enough detail to support parallel work by separate teams developing different system modules. For example, the language allows designers to write the (un)expected join points and to define the responsibilities of both class and aspect developers. These requirements are checked statically by an extended AspectJ compiler that we have developed. For instance, if a developer needs to create an aspect `A` (see Listing 1.1) that depends on (intercepts) a call to a method `m2` within method `m1`, our

language can express this design rule and check, during compilation, if the required call occurs. If the call is missing (as illustrated in Listing 1.2), our tool reports the error back to the developer.

We compared the specification of design rules from the Health Watcher system [90] using both XPIs and LSD, and noticed that the design rules expressed in LSD were more expressive and concise than XPIs. Beyond all the advantages cited so far, the usage of a specific language for this purpose brings the following benefits:

- Description of Design Rules in a simple and unambiguous manner, enabling automatic checking of rules against code.
- Support for the creation of a guideline to be utilized since the initial development phases of the components, specifying the essential constructions so that each developed component can have the proper functionality;
- A declarative notation that is easy to read and write and not very different from AspectJ.

1.2.1 Hypothesis

The use of a language that was designed with the sole purpose of specifying design rules, with a clearly defined semantics and expressive enough to specify most of the design rules in AO systems, improves both class and crosscutting modularity, when compared to an oblivious approach, but does not present the problems of informal design rules and XPIs.

1.2.2 Summary of Contributions

Accordingly, the main contributions of our work are:

- The definition of a language for specifying design rules [72, 31, 70], improving the modularity of AO systems, in a declarative manner, using a syntax similar to Java [40] and AspectJ [51], and with a formal semantics (Chapter 3).
- A specification of the language semantics [72] in Alloy [49] which allows others to implement tools supporting LSD. As far as we know, this is the first attempt to formally define a language for specifying design rules (Chapter 4).
- An evaluation of the proposed language using the Health Watcher system [90] as case study and the comparison with other design rules-based approaches (Chapter 6).
- Development of a tool to support the introduction of design rules in the development process of AO systems (Chapter 5).

1.2.3 Relevance and Applications

AOP in its present form, in spite of improving crosscutting modularity, breaks class modularity, requiring that developers take aspects into consideration when reasoning about classes in the system. Our approach restores the class modularity and preserves the crosscutting modularity introduced by AOP. These benefits that stem from our approach enable the use of AOP in large and complex applications and specially in Software Product Lines (SPL). A SPL is a set of software-intensive systems sharing a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way [21]. SPL can be implemented with AOP languages like AspectJ [4, 5, 6, 26, 25] but we can control dependencies between classes (*e.g.*, expected method calls, required method inter-type declarations) and aspects (implementing the variabilities) more clearly with Design Rules.

On the other hand, there are some basic aspect implementations that are so general that make no explicit reference to classes or interfaces (no coupling). This is the case in aspect-oriented frameworks [89], which include reusable implementations of several concerns that can be specialized as required. Moreover, some aspects are inherently simple and do not require their interaction with classes and aspects to be specified. This is the case with development aspects, such as the ones implementing tracing. LSD does not impose restrictions that constrain the development of aspects in these situations, since aspects are not required to implement design rules.

An important question that arises when we assume that developers should establish design rules between classes and aspects is if we are compromising or not the *obliviousness* characteristic expected from AOP, and consequently limiting its usefulness. But, as there are several definitions and different degrees of obliviousness (as discussed by Sullivan [94]), we summarize some of them below:

Language-level obliviousness: Class developers do not need to use any type of signal or explicit notation (callback hooks, annotations or macros) within classes in order to activate aspects.

Feature obliviousness: Class developers design the code exposing event hooks (join points) in accordance with aspect developers, but still unaware of the features that aspects implement.

Designer obliviousness: Class developers are completely oblivious to the existence of aspects, designing in the same way as they normally would in the absence of aspects.

Our approach does not provide *designer obliviousness* because developers usually design classes and aspects differently to facilitate the interaction between classes and aspects. However, both *language-level* and *feature obliviousness* are met by our solution. We do not see this as a limitation because we and other authors [94] argue that AO software developed under oblivious designer conditions frequently results in complex and highly coupled software. Designer obliviousness discourages the use of AOP to implement software with numerous aspects or in naturally complex software.

1.3 Organization

In the following chapter, we discuss the modularity problems in AO systems and present a survey about the state of the art on Software Modularity of AO Systems, discussing their relation with our work. In Chapter 3, we present our language for specifying design rules in AO Systems. Chapter 4 shows the formal specification of the language in Alloy. We present, in Chapter 5, details about the extension of an AspectJ compiler to support LSD constructs. We evaluate our language using the Health Watcher system [90] as case study and compare it with existing approaches in Chapter 6. In Chapter 7, we summarize the benefits, contributions, assumptions and limitations of our work. Also, we discuss related and future work. Appendix A presents more details about the formal specification of the language in Alloy (partially presented in Chapter 4). Finally, we show the BNF specification of LSD in Appendix B.

Chapter 2

State of the Art on Software Modularity of AO Systems

In this chapter, we present the concept of modularity we adopt and show how modularity can be evaluated through Design Structure Matrixes (Section 2.1). Then we present modularity issues in Aspect-Oriented Programming (Section 2.2). Following this, we discuss several works that contribute to solve the AO modularity problems (Section 2.3).

2.1 Software Modularity

The criteria we use in this work for considering a modular design is based on the ideas of David Parnas [78]. Although presented in the early 1970s, this criteria is still used as a guide for architects and has been applied in other areas. His notion of modularity is closely related to design decisions that decompose and organize the system into a set of modules. Moreover, the following qualities attributes are expected in a modular design:

Comprehensibility: A modular design allows developers to understand a module looking only at: (1) the implementation of the module itself; and (2) the interfaces of the other modules referenced by it.

Changeability: A modular design enables local changes. If changes are necessary in the internal implementation of a module A, the other modules that depend exclusively on *A's interface* will not change, since there is no modification in the module interface.

Parallel development: After the specification of the module interfaces, a modular design enables the parallel development of modules. Different teams can focus on their own modules development, reducing time-to-market and communication needs.

Besides that, Parnas proposed the *information hiding* principle as the criteria to be used when decomposing systems into modules. According to Parnas, the parts of a system that are more likely to change must be hidden into modules with stable interfaces.

Reinforcing these ideas, Baldwin and Clark [12] proposed a theory which considers modularity as a key factor to innovation and market growth, independent of industries

domains. Their theory uses Design Structure Matrixes (DSMs) [92] to reason about dependencies among artifacts and argues that the task structure of an organization is closely related to such dependencies. As a consequence, if two modules are coupled, they cannot be developed in parallel, which actually requires either (a) more communication between different teams; or (b) their implementation to be assigned to a single team.

Following that, in this work we use Design Structure Matrixes (DSMs) as a tool for visualizing dependencies among design parameters. These parameters correspond to decisions that need to be made along product design. Design parameters might have different abstraction levels. In the software industry, for example, some design decisions are related to development process, programing language choice, code or architectural style [84]. Moreover, here we also consider implementation as a design activity. Therefore, classes, interfaces, packages, and aspects are also represented as design parameters, as we need to make several decisions about them.

A dependency arises whenever a design decision depends on another. In a DSM, a dependency between two design decisions (or parameters) is represented with an "x" in the corresponding lines and columns. For example, suppose that the DSM depicted in Figure 2.1 represents software components as parameters. The mark in row B, column A indicates that design decisions regarding component B depend on decisions concerned to component A. In a similar way, an "x" in row A, column B indicates that design decisions of component A depend on decisions of component B. Whenever this mutual dependency occurs, we have a cyclical dependency, which in fact implies that both components cannot be independently addressed. As a consequence, their parallel development is compromised.

	A	B	C
A		x	
B	x		x
C			

Figure 2.1: Example of dependencies in a DSM.

In the same DSM of Figure 2.1, component B depends on C (expressed by an "x" in row B, column C) but C does not depend on any other component (there is no mark in row C). Therefore, C can be independently developed but B cannot be completely developed until design decisions of C have been established. We can improve modularity by removing dependencies between design decisions, but several assumptions must be made before that. Such assumptions, represented as a special kind of parameter, are named, by Baldwin and Clark [12], as Design Rules.

Therefore, Design Rules are parameters that are less likely to change and are used as interfaces between modules [65]. In this way, they are used to decouple design parameters, like typical programming interfaces remove the coupling between software components. Such design rules establish strict partitions of knowledge and effort at the outset of a design process. They are not just guidelines or recommendations: they must be rigorously obeyed in all phases of design and production [12]. Figure 2.2 illustrates components A and B being decoupled by design rules (DRs). Since A and B do not depend on each other anymore, it is possible, for example, to change A's implementation

as long as it respects the design rule. In addition, when both respect the design rules, parallel development becomes possible [71]. However, notice that the coupling does not disappear, but it is better managed. Its place has been changed instead: A is coupled to the design rule. The same happens to B.

	DRs	A	B	C
DRs				
A	x			
B	x			x
C				

Figure 2.2: Design Rules decoupling components A and B.

2.2 Modularity Issues in Aspect-Oriented Programming

Aspect-Oriented Programming was proposed to modularize crosscutting concerns [52]. However, constructions supported by AspectJ-like languages can produce high coupling between classes and aspects, which may compromise the criteria discussed previously. In this section, we illustrate this problem with some examples, extracted from the Health Watcher system [90].

Health Watcher (HW) is a real web-based information system originally implemented in Java and later restructured with AspectJ [51]. The system was developed to improve the quality of the services provided by health care institutions, allowing citizens to register complaints regarding health issues, so that health care institutions can investigate and take required actions. We have selected the HW system because its design has a significant number of non-crosscutting and crosscutting concerns. Furthermore, it requires a number of recurrent design decisions related to GUI, persistence, and concurrency concerns. Finally, the HW system has been used as the case study of several *aspect-oriented* works [41, 42, 90]. Figure 2.3 shows the core architecture of the HW system. This architecture aims to modularize the user interface, distribution, business rules, and data management concerns. Next we describe the major architectural components of the HW system:

View Layer: related to the HW web interface. The implementation of this layer is based on the Front Controller [3] and Command [38] patterns, using *servlets* and *plain Java objects*. The communication with the business layer is implemented by means of calls to the interface `IFBusiness`, which may be distributed or not.

Business Layer: responsible for implementing both the business logic and transactional concerns. The class `HWFacade`, which implements `IFBusiness`, is the unique point of interaction with this layer and is based on the Facade pattern [38]. This class uses `record` components to interact with the Data Access Layer.

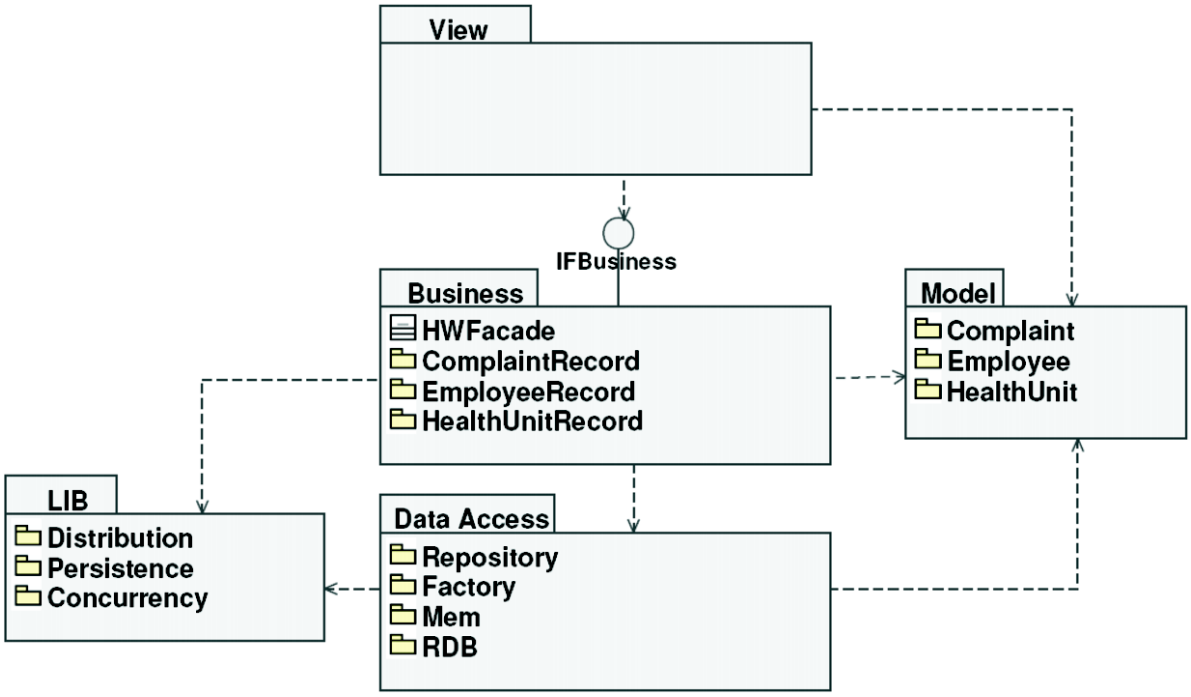


Figure 2.3: Overview of the Health Watcher architecture (OO version).

Data Access Layer: responsible for abstracting the persistence mechanism following the Data Access Object pattern [3]. Some interfaces to manage data persistence are defined in this layer. Two implementations are available: the first one uses volatile memory whereas the second one is based on relational databases.

Model: responsible for implementing the *domain objects*. These objects represent the core concepts of the application, transit between all architectural layers, and have some business logic. *Complaint*, *employee*, and *health unit* are examples of core concepts in the HW system.

Lib Components: represent reusable components that are useful for the implementation of concerns like persistence, distribution, and concurrency.

By analyzing this system, we present some issues related to *aspect-oriented* modularity. First, we discuss the *fragile pointcut* problem in Section 2.2.1. After that, we argue that clear interfaces, for decoupling crosscutting code related to different concerns, are necessary for developing different concerns in parallel and supporting software maintenance activities.

2.2.1 Fragile Pointcut problem

The first issue mentioned above can be observed with the implementation of the transaction management concern, which affects the behavior of certain methods of the HW business layer. The implementation of this concern consists of adding transactional behavior, using `begin` and `commit` or `rollback` commands. Listing 2.1 illustrates a piece of the source code related to this concern in AspectJ.

Listing 2.1: Aspect responsible for implementing the transactional concern.

```
1  public aspect TransactionAspect {
2
3      public pointcut transactionalMethods() : /* ... */
4
5      before() : transactionalMethods() {
6          transMechanism.begin();
7      }
8      after() returning : transactionalMethods() {
9          transMechanism.commit();
10     }
11     after() throwing : transactionalMethods() {
12         transMechanism.rollback();
13     }
14
15 }
```

If a class developer is oblivious about the `TransactionAspect` and decides to implement a new transactional method (`newMethod`), at least two problems might occur:

1. If `newMethod` should be transactional but the transactional management is not implemented within it and its join points are not matched by the `transactionalMethods` pointcut, the method will not work as expected. This situation is relatively common because class developers might be oblivious about the pointcut expression defined by aspect developers;

2. If the class developer implements itself the transactional management concern in the new method (explicitly calling `transactionMechanism.begin`), and this method is coincidentally matched by the `transactionalMethods` pointcut, an unexpected behavior occurs, since two consecutive calls to *begin transaction* are made.

In both cases, we need to change the pointcut in order to capture (or not) the join points. That is why this problem is called *fragile pointcut* [93]. The situation above exposes some modularity problems: (1) *comprehensibility is compromised*, because two modules should be studied in order to understand and correctly implement the concern; and (2) *parallel development is problematic*, because it is impossible to implement classes and aspects independently and be sure that one does not interfere on the other. Also, one developer can implement unintended behavior into a module which, although under his responsibility, may break other parts of the system. Moreover developers might need to know details about other modules in order to implement them correctly. This is true even to modules that are not explicitly used by the concerns implemented as aspects.

So any unanticipated change in the class might cause problems and the application may not behave as presumed. Note that we have a cyclical dependency in this situation: the aspect depends on the class syntactically (notice that pointcut expressions refer to classes, interfaces, aspects and their members); and to change the class, the developer must be aware of the aspect. Figure 2.4(a) illustrates such a cyclical dependency through a DSM, whereas Figure 2.4(b) shows design rules coming into play to remove the dependency between aspect and class.

In summary, this problem indicates that we need interfaces (or design rules) that delimit the scopes from which it is allowed to call certain methods (in this case `begin`, `commit`, and `rollback` within `TransactionAspect`) and expose the expected method

		1	2
TransactionAspect	1		x
Business Classes	2	x	

(a) Cyclical Dependency.

		1	2	3
Transaction Management DR	1			
TransactionAspect	2	x		
Business Classes	3	x		

(b) Cyclical Dependency removed.

Figure 2.4: Transaction concern without and with Design Rules.

signatures to be matched by the `transactionalMethods` pointcut. Through the establishment of these design rules, both class and aspect developers can work more independently because the points of interaction between them are defined a priori [94]. If they respect the constraints, many defects, delays and problems can be avoided.

2.2.2 Unsupported Parallel Development

Another kind of modularity issue may arise when a team is assigned to develop a crosscutting concern and another team is assigned to develop a non-crosscutting concern. This happens, for instance, with the transaction management concern from Listing 2.1 because aspect developers need to know the transactional points and the business layer class developer needs to be conscious about the aspect. Without a clear interface between those concerns, substantial communication might be required, which, in fact, compromises parallel development of both concerns and causes the fragile pointcut problem during system maintenance. This is aligned with Parnas [78], and Baldwin and Clark [12] observations that modularity is related to the assignment of development activities, which may be reflected in the program structure [30], and that a modular design reduces the communication paths among design decisions, in such a way that units of work can be developed in parallel.

For instance, suppose that a team is responsible for developing the use case related to the *complaint management* concern (a core concern of the Health Watcher system). And another team is responsible for an auditing concern that must be triggered whenever a change in a complaint occurs. Without a clear interface stating which are the relevant complaint changes (the set of joinpoints) and how these joinpoints should be written by the complaint management team, any increment in the use case must be communicated to the auditing team. Consequently, it is difficult to implement the auditing concern at the same time that the complaint management concern is being developed. Although these concerns can be encapsulated in single code unities (locality), there is no modular design in this case because the unities need to know implementation details about each other.

To better explore this, consider the DSM depicted in Figure 2.5, which represents some design parameters and respective dependencies of the HW system. Based on this DSM, we can realize that decisions about the complaint implementation (row 5) depend on decisions about complaint requirements (dependency row 5, column 2), architectural decisions[1] (dependency row 5, column 4), and the auditing concern (dependency row

[1]Examples of architectural decisions for the Health Watcher system are the selected architectural style (layers), patterns, and technologies for each layer or concern (presentation, distribution, persis-

5, column 6). This last dependency exists because the team responsible for developing the *complaint concern* has to know which joinpoints must be exposed to the *auditing concern*. Moreover, as we can observe in Figure 2.5, the auditing implementation also depends on the complaint implementation decisions, since changes in its implementation should be notified to the auditing implementation team. In this way, there is a cyclical dependency between complaint implementation and auditing implementation — a clear example of non-modular design.

		1	2	3	4	5	6
Goals and constraints	1	■					
Complaint requirements	2	x	■				
Auditing requirements	3	x		■			
Architectural decisions	4	x	x		■		
Complaint implementation	5		x		x	■	x
Auditing AO implementation	6			x		x	■

Figure 2.5: Cyclical dependency between Complaint and Auditing

Based on the information hiding principle, we should encapsulate the dependencies between complaint and auditing concerns in a special kind of interface (a design rule). Applying design rules to this example, we improve the design structure by removing the cyclical dependencies between the complaint and auditing concerns. The new DSM is presented in Figure 2.6. Notice that a new parameter (actually a design rule) was introduced (row 5) and all dependencies are below the main diagonal (there is no more cyclical dependencies). This design rule was proposed in order to improve the parallel development between class developers and aspect developers. In fact, it is responsible for establishing contracts that define what is expected from both teams. In this case, among other things, this design rule establishes that class developers must follow specific naming conventions for methods that represent auditing points. Based on that, aspect developers can implement the auditing concern.

		1	2	3	4	5	6	7
Goals and constraints	1	■						
Complaint requirements	2	x	■					
Auditing requirements	3	x		■				
Architectural decisions	4	x	x		■			
Auditing Design Rule	5		x	x		■		
Complaint implementation	6		x		x	x	■	
Auditing AO implementation	7			x		x		■

Figure 2.6: Auditing design rule removes cyclical dependency

Our approach addresses these problems through the use of a language to express and enforce design rules to both classes, interfaces and aspects. This language is presented in Chapter 3.

tence, and so on).

2.3 Current Approaches for Improving Modularity in AO Systems

AO language designers can promote other software engineering concerns, like class modularity, while only giving up a small amount of what makes them aspect-oriented. Put it another way, AO languages can promote increased locality (*i.e.*, reduced scattering) and separation of concerns (*i.e.*, reducing tangling), while still promoting ease of maintenance and ease of reasoning about classes. By keeping in mind that all these concerns can be compromised in small amounts, language designers can find creative ways to simultaneously promote them all [62].

In this section we discuss some works that contribute to evaluate software modularity and to solve the AO modularity problems, some proposing extensions to existing AO languages, some checking constraints in AO programs, others trying to increase the abstraction level of pointcut language and even languages that resemble AO languages. We also present some AO applications that can benefit from modularity improvements, like AO Software Product Lines.

2.3.1 Software Modularity Evaluation

Analyzing a software to determine its quality requires a well defined approach, capable of considering several software perspectives like coupling, cohesion and complexity. We discuss below some works related to software modularity evaluation separated in two groups. The first group is based on metrics and the second uses design structure matrixes (DSMs). None of them solve the modularity problem, but help to identify and quantify the dependencies. We use DSMs to visualize the dependencies between classes and aspects, but we do not use metrics to quantify them, mainly because most of the dependencies we discuss do not have a corresponding metrics yet.

Metrics

Zhao uses the concept of dependence graphs that represents various dependency relations in a program to create a dependence model for AO software. Based on this model, he proposed methods to assess the complexity [106], coupling [107], and cohesion [109]. There are other works that are more focused on measuring coupling [17, 13] because AOP introduces new kinds of coupling when compared to OOP.

Sant'Anna *et al.* [87] presents an assessment framework for AO software, as suggested by Zakaria and Hosny [105], consisting of an extension to the metrics suite (known as CK metrics) proposed by Chidamber and Kemerer [20]. They proposed some metrics for measuring separation of concerns (useful in the AO context) and reviewed size, coupling, and cohesion metrics to consider aspects. Subsequent work proposes the use of an assessment tool [36] and evaluates their framework in practical case studies, like the GoF Design Patterns [39]. These metrics are also used in a quantitative case study that assesses and contrasts the design stability of OO and AO designs for 10 releases of Health Watcher considering a number of system changes that are typically performed during software maintenance tasks [41].

Design Structure Matrixes

Design (or Dependency) Structure Matrixes (DSMs) (discussed in Section 2.1) are specially useful to visualize the dependencies between components, tasks and many other kinds of artifacts, as demonstrated in Section 2.2.

Sullivan *et al.* first demonstrated how the methodology using DSMs and Net Option Value (NOV) can be used in the analysis of software design [95]. NOV is a model for evaluating modular design structures based on the economic theory of real options. Baldwin and Clark [12] formulated NOV and first demonstrated its usage in analyzing design options in the computer hardware industry. There are two fundamental components in Baldwin and Clark's work:

- A general theory of modularity in design with six modular operators[2] as sources of design variation;
- NOV as a mathematical model to quantify the value of a modular design: the mathematical expressions for NOV tie together modular dependencies, uncertainty, and economic theory in a cohesive model [65].

Using NOV analysis, Sullivan *et al.* [94] showed how information hiding design is superior to the protomodular one. Information hiding is achieved by defining appropriate interfaces as Design Rules, which facilitate future changes in the design by reducing inter-modular dependencies.

Lattix LDM [48, 86] is a tool that supports the definition of design rules (*e.g.*, a class from package X cannot depend on classes from package Y) that are checked repeatedly against the code as the system evolves, aiming to identify violations, and keeping the systems in conformance with the design rules. The tool can build DSMs based on some languages. It is important to notice that they use the term *design rules* with a different meaning, which is basically the existence (or not) of some dependency between two elements.

2.3.2 Architectural and Model-Based Approaches

Some works defend that aspects should be identified since the initial phases of the development process, clearly defining their requirements and how they interact with other components. This thesis agrees with that. Assuming that the system documentation is kept updated with the system, developers can use this documentation to understand existing aspects, their dependencies with other components and identify which of them may be affected by the changes the developer plans to do.

Chavez *et al* [18] presented crosscutting interfaces as a conceptual tool for dealing with the complexity of heterogeneous aspects at the design level. This work also presents a modeling notation for describing architectural-level aspects that also supports the explicit representation of *crosscutting interfaces*. However, although using a visual notation is important for documentation purposes, they do not enable to check if the code was built according to established interfaces because they do not have any

[2] (1) Splitting, (2) Substitution, (3) Augmentation, (4) Exclusion, (5) Inversion, and (6) Port(ing).

representation at implementation level. Our proposal, besides enabling the specification of a major number of design rules not supported by the visual notation, allows to check automatically if code is in conformity with the specified design rules.

One interesting challenge regarding the pragmatics of our approach is how to discover stable design rules for decoupling aspect and class development. In this context, Landuyt et al. present a process for identifying abstractions that lead to reusable pointcut signatures [97]. Their idea follows a top-down approach, where relevant crosscutting concerns are identified and reasoned about throughout the architecture design. Using their approach, developers should be able to "build pointcut interfaces that are resilient to evolution" [97]. Although this concept of resilient interfaces is very close to our notion of design rules, the interfaces discussed in [97] mainly expose stable joinpoints. In fact, this kind of interface could mitigate the *fragile pointcut* problem, but they are not expressive enough to clearly present which are the obligations of class and aspect developers. For instance, using the mentioned approach, architects could not be able to find that calls to a specific method have to occur within a specific advice. It is a matter of future work to adapt their approach to find the other kinds of interfaces discussed in our work.

Kellens *et al.* [50] propose declaring pointcuts in terms of a conceptual model of the base program, rather than defining them directly in terms of how the base program is structured. As such, they achieve a more effective decoupling of the pointcuts from the base program's structure. In addition, the conceptual model provides a means to check where and why potential fragile pointcut conflicts occur, by imposing structural and semantic constraints on the conceptual model, that can be checked when the base program evolves. CrossMDA2 [27] mitigates this problem through the adoption of tools and modeling languages that are usually used by software engineers, like Unified Modeling Language (UML) [15] and Object Constraint Language (OCL). However, both approaches require additional models and development process adaptations.

Kulesza [55] proposes a systematic approach to framework development which relies on the use of aspect-oriented (AO) techniques. The main goal of the approach is to improve the extensibility and configurability of object-oriented (OO) frameworks. It defines a set of guidelines to design and implement frameworks using AO programming, that include the definition of extension join points (EJPs) in the framework code. Similar to XPIs [94, 43], EJPs establish a contract between the framework classes and a set of aspects extending the framework functionality. Unlike XPIs, however, EJPs aims at increasing the framework variability and integrability [57]. EJPs can be used to extend the framework basic functionality by means of extension aspects. As Figure 2.7 illustrates, EJPs are used by the extension aspects, that are responsible for implementing optional, alternative and integration crosscutting features required by the framework users. Since such aspects can be automatically unplugged from the framework code, the approach makes it easier to customize the framework to specific needs. In addition to that, it was built a systematic approach, based on EJPs, for testing systems that have many of their crosscutting features implemented by means of aspects [23]. In the same way that XPIs, EJPs can benefit from a language for specifying design rules, like ours. The approach also introduces a model-based generative model which allows the automatic instantiation of the framework and its respective OO and AO variabilities [56]. This model can be adapted to use design rules expressed in our language.

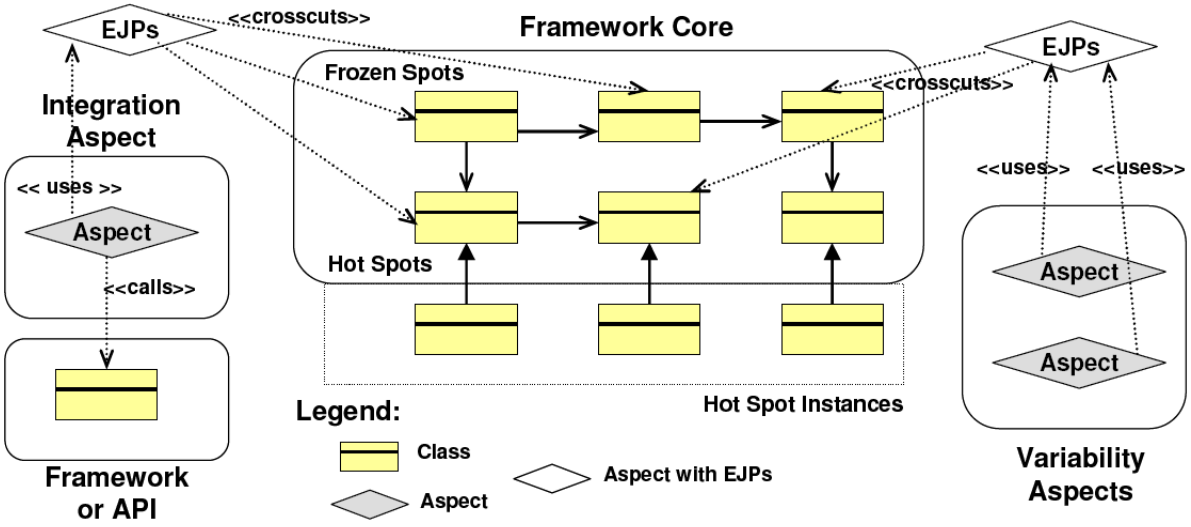


Figure 2.7: Framework Development Approach based on EJPs.

2.3.3 Refactoring of AO Programs

Refactoring is defined by Fowler *et al.* [37] as "a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior". AOP requires special attention when performing refactorings because whenever we change a class, aspects may be affected. Also, it is possible to transfer behavior from classes to aspects (*e.g.*, transform a method in an advice).

Cole and Borba [24] present a set of aspect-oriented programming laws that can be used for deriving refactorings for AspectJ. These refactorings support the safe code migration from classes to aspects.

In summary, refactorings are useful to give support to developers during maintenance activities of AO programs, including refactorings in non-modular programs. Without this support, it is difficult to cope with existent dependencies between program elements. However, refactorings require that all existing code from a program is available so that conditions can be checked and all affected components are updated. We argue that with a broader interface (design rules) between classes and aspects, dependencies between them are documented, supporting changes to hidden parts of a program in a safer manner. Without DRs, changes to hidden parts of a class are not safe because, in contrast to an OO program, aspects can depend on these parts and these dependencies are not explicitly stated.

2.3.4 Disciplined use of AOP

Some researchers defend that if AOP is used in a disciplined form, it is possible to prevent many of the presented problems. On the other hand, this disciplined form may impose limitations that restrict the use of AOP in such way that it might become useless or at least limited. We advocate that, by using design rules, we can deal with modularity problems despite of disrespecting some of the design principles presented in these works.

Wampler [101] advocates that appropriate design principles are needed to guide AOP

use in real, evolving systems. The principles should tell us what types of coupling are appropriate between aspects and the software entities they advise, how to use non-invasiveness effectively, how to preserve correct behavior in the advised entities, and how to use aspects with other design constructs. Object-Oriented design principles are considered from an AO design perspective, and it shows how AO design contributes to design solutions that satisfy these principles. Additionally, some AO design specific principles are derived from the OO design principles.

Some approaches classify aspects with respect to their effect over the classes and aspects (*e.g.*, pure behavioral, pure data and hybrid aspects) [69] and how they interact with others aspects (*e.g.*, mutual exclusion, dependency, reinforcement and conflict) [85]. These classifications help to reduce the scope of analysis when we are looking for errors and also to better understand aspects interaction.

Those works are important because guide developers to create well-designed AO programs, and provide information about the interference of existent aspects on classes to developers, making easier to reason about the program. However, we argue that with DRs we can provide more details about the interaction between classes and aspects, without imposing unnecessary limitations to aspects.

2.3.5 Design by Contract

Design by Contract (DbC) [66, 67] helps to improve the modularity of software because we can specify more constraints than an expected set of public methods. Through contract specifications, we can implement aspects and modify classes with more confidence. However, these contracts are specified inside each component and not in a separate interface implemented by the component that can be shared by other components. With AOP this limitation is even more problematic because aspects can intercept and depend on the structure and behavior that goes beyond typical public interfaces. Besides, these contracts specify constraints that are checked during runtime. However, it would be interesting to check some of these constraints at compilation time.

Java Modeling Language (JML) [59, 61, 63] is a behavioral interface specification language tailored to Java. Besides pre- and postconditions, it also allows assertions to be intermixed with Java code; these aid verification and debugging. Listing 2.2 illustrates how to avoid division by zero through a JML specification containing a precondition clause, requiring b $>$ 0. This precondition states that the input parameter b must be grater than zero. JML is designed to be used by working software engineers; to do this it follows Eiffel in using Java expressions in assertions, but provides expressiveness advantages over Eiffel which include quantifiers, specification-only variables, and frame conditions.

Listing 2.2: JML Example.

```
public class Math {
  //@ requires b > 0;
  public int div (int a, int b) {
    return a/b;
  }
}
```

A recent work [80, 82, 83, 81] presented a new JML compiler – ajmlc (AspectJ JML Compiler) – that uses AspectJ to instrument Java code with JML predicates. A set

of translation rules are defined from JML predicates into AspectJ program code. The translation rules handle a number of JML specifications, such as pre-, postconditions, and invariants.

Similarly, Contract4J [100] supports DbC in Java using AspectJ for the implementation. It demonstrates an alternative approach to aspect-aware interface design with two syntax forms supporting a common protocol. Instead of specifying methods to implement, like a traditional Java interface, one form uses Java 5 annotations and the other uses a method naming convention. Both approaches are essentially a design pattern [38] that must be followed by components that wish to expose key information, in this case a usage contract, and by clients interested in the usage contract, which they can "read" from the components if they understand the protocol, without having to know other details about the components. Compared to JML and the ajmlc compiler [80], this tool has similar purposes but is more limited with respect to the set of constraints that can be specified.

Pipa [108] is a behavioral interface specification language (BISL) tailored to AspectJ. Pipa is a simple and practical extension to JML [59]. Pipa uses the same basic approach as JML to specify AspectJ classes and interfaces, and extends JML, with just a few new constructs, to specify AspectJ aspects. Pipa also supports aspect specification inheritance and crosscutting. They show how to transform an AspectJ program together with its Pipa specification into a corresponding Java program and JML specification. The goal is to facilitate the use of existing JML-based tools to verify AspectJ programs.

Although existing approaches to DbC do not completely solve the modularity problems, they can be very useful in conjunction with other approaches, like ours. In fact, one of our future works is to include some DbC features to our language, enabling the checking of some design rules at execution time.

2.3.6 Design Rule Specification Languages and Checking Tools

Some works support the specification and checking of design rules in OO programs using declarative query languages [98, 68] to express design violations or APIs [16] that supports the development of programs that analyze the code and check the design rules. However, these approaches do not support aspects and, most importantly, are focused on prohibiting undesirable design instead of guiding developers throughout program construction. It is important to notice that these languages use the term *design rules* to designate a set of conditions that are checked against that source code. Each approach uses a different mechanism to perform this checking, but differently from our approach, their design rules are not useful neither to guide developers nor to decouple classes from aspects.

SemmleCode [98] is an Eclipse plug-in that supports tasks like navigating code, finding bugs, computing metrics, checking style rules, and enforcing coding conventions, through queries over the code base. A library of queries for common operations is provided, including metrics. Query results can be displayed as a tree/table view, in the problem view, as charts or graphs, all with links to the source code. To illustrate, consider the problem of ensuring consistency between `compareTo` and `equals`. It makes sense to indicate a warning whenever `compareTo` is defined by a class, but `equals` is not. In SemmleCode, it is possible to generate a warning with the query shown in Listing 2.3

Listing 2.3: Semmle Code Example.

```
from Method m
where m.hasName("compareTo") and
   not(m.getDeclaringType().declaresMethod("equals"))
select m, "Is compareTo consistent with equals?"
```

Design Wizard [16] is based on the concept of *design tests*, which are test-like programs that automatically check whether an implementation conforms to a specific design rule. Design rules are implemented directly in the target programming language in the form of tests. Design Wizard provides an API developed to support design tests for Java programs as JUnit test cases. In summary, developers can write constraints using the available API and run design and unit tests together. As an example, suppose that we need to check whether the classes inside the `dao` package are only called by the classes inside the `controller` package of the same component. In other words, any attribute access or method call regarding the `dao` package classes must come from the `controller` package or from the `dao` package itself. Checking this rule avoids unnecessary direct coupling between presentation objects and data objects, making the code easier to modify and maintain. A corresponding design test for that is presented in Listing 2.4.

Listing 2.4: Design Wizard Example.

```
public class OurGridDesignTest extends TestCase {
   public void testCommunication() {
      DesignWizard dw;
      dw = new DesignWizard("ourgrid.jar");
      PackageNode dao = dw.getPackage("org.ourgrid.peer.dao");
      PackageNode controller = dw.getPackage("org.ourgrid.peer.controller");
      Set<ClassNode> callers = null;
      for (ClassNode clazz : dao.getAllClasses()) {
         callers = clazz.getCallers();
         for (ClassNode caller : callers) {
            assertTrue(caller.getPackage().equals(dao) ||
                  caller.getPackage().equals(controller))
      }
   }
}
```

Program Description Logic (PDL) [68], inspired by the pointcut language in AspectJ, allows succinct declarative definitions of programmatic structures which correspond to design rule violations. They express undesirable declarations in code, *e.g.*, public attributes, calls from certain places and inheritance declarations. The general idea is explicitly declaring what is prohibited and analyze the program to check if something is not obeyed.

Dependency Constraint Language (DCL) [99] is a domain specific language that supports the definition of structural constraints between modules, restricting the set of allowed dependencies in OO systems. DCL supports the definition of modules and constraints. Modules can range from a single class to all classes from an external library. DCL constraints are formed by a quantifier (*e.g.*, `can-only`, `cannot` and `must` associated to a dependency type (*e.g.*, `access`, `declare`, `create`, `extend` and `implement`) that describes the dependencies between modules. It provides a tool (*dclcheck*) to check if a given system satisfies the constraints. *Lattix LDM* [48] provides similar functionality but with less expressivity power.

SemmleCode, *Design Wizard*, *PDL*, *DCL* and *Lattix LDM* in their current versions are useful only in the development of OO systems because they do not support aspects.

Besides, they do not provide any extension to the traditional interface notion of Java. We argue that it is necessary to expand the interface notion between classes and aspects through a new language. In spite of that, those languages can serve as an infrastructure to a static analyzer that checks if a program is in conformity with its design rules, as long as they were extended to support AOP.

2.3.7 Aspect-Oriented Language Extensions and Related Approaches

In this section we present some AO language extensions that aim to improve some language feature, like a more expressive pointcut language or restrictions to join points access. Also, we discuss some works that present similarity with AO approaches and bring interesting solutions to AOP modularity problems.

Open Modules [2, 75] propose the use of interfaces for exposing *join points* in the classes, limiting the scope of advised code to the join points exposed by the class developer which, in fact, defines the responsibilities just for class developers. Moreover, this approach does not prevent the fragile pointcut problem because class developers might remove expected join point shadows, leading aspects to misbehave. However, as pointcuts are under class developers responsibility it may be easier to detect the problem. Therefore, this approach does not offer mechanisms for describing the responsibilities of aspect developers. As a consequence, class developers are still able to implement part of a concern assigned for a different team but they cannot assume the existence of any behavior expected to be modularized as an aspect.

Larochele *et al.* [58] have proposed a mechanism, called *Join Point Encapsulation*, which aims to prevent selected join points from being intercepted by aspects. They extend the AspectJ language to support a *restrict advice* that prohibits the interception of specific join points, as shown in Listing 2.5. The pointcut `private_impl` defines a set of join points that are restricted by a restrict advice. This advice can be declared in any aspect, including inner aspects, allowing each class to specify what restrictions apply to it.

Listing 2.5: Join Point Encapsulation Example.

```
public aspect SomeAspect {
   pointcut private_impl() : call (private * SomeClass.*_impl(..));
   restrict() : private_impl();
}
```

Confirmed Join Points [76] is an approach that tries to balance the search for control by class developers and the flexibility expected by aspect developers. With confirmed joint points, pointcuts can be written and modified at will by pointcut owners (who might also be aspect owners or class owners). However, class owners must confirm explicitly that join points matching those pointcuts are acceptable, and those confirmations remain visible in the class code. In summary, both class and aspect developers must be aware of join points, causing the tangling and scattering of join point confirmations.

Ptolemy [79] is a programming language that provides quantified, typed events. It joins ideas from Implicit Invocation (II) and AO programming languages. Implicit invocation languages have explicitly announced events, which runs registered observer methods. AO languages have implicitly announced events, called join points which

dispatch advice executions. Ptolemy supports the declaration of event type, which have a name that can be used in quantification. An event type also declares the types of information communicated between events and observer methods for events of its type, partially performing the role of an interface. Events are declaratively identified using event expressions that name the event's type. As events are scattered and tangled throughout the code and it is possible to tell when an "advice" will execute, Ptolemy does not provide *language-level obliviousness* because developers use explicit signals to activate aspects. However, it provides *feature obliviousness* because developers are still unaware of the features that aspect implement. The language design is closer to II languages than AO languages.

Another approach, named *Explicit Join Points (EJP)* [47], explores new possibilities in the language design space that opens up when we assume that classes and aspects are aware of crosscutting aspects. EJP serves as an explicit representation of an abstract contract between classes and aspects, modeling the information required by the crosscutting concern and also any constraints to be enforced upon classes wishing to be advised. In contrast to method invocation, however, the target class is not predefined, and the classes are not coupled to any specific implementation. Both EJP and Ptolemy share the common idea that classes must explicitly know the join points that are exposed to aspects and its constraints (context), but EJP is closer to AspectJ design while Ptolemy resembles Implicit Invocation languages.

Expressive Pointcuts [77] introduces a new pointcut language in the form of logic queries over different models of the program semantics. Together with the abstraction facilities of logic programming, it becomes possible to raise the abstraction level of pointcuts and hence increase the software quality of aspect-oriented code. It provides a way of specifying pointcuts that express "when" (under which conditions) instead of "where" (lexically) an advice must be applied. It presents a static analysis technique that can be the starting point of an efficient implementation, but they do not provide real compiled programming language with the new pointcut language. Although this approach increases the abstraction level of pointcuts, we believe that the establishment of design rules can be useful to improve the modularity because class developers may still depend on aspects to work correctly.

PointcutDoctor [103, 104] provides several kinds of information for a given pointcut. Firstly it shows which join points (shadows) the pointcut matches or does not match. This helps a developer to check whether the pointcut is correct. Furthermore, PointcutDoctor also provides an explanation of why the pointcut matches or does not match a given join point (shadow). This information helps a developer diagnose the cause of problems – unintended matches or failures to match certain join points – in the pointcut. The main problem of this approach is that it requires that the system as a whole is available to analyze its join points and detect problems. Also, it does not support system modularity improvement, since it does not provide mechanisms to hide information or define interfaces between classes and aspects.

2.3.8 Software Product Lines implemented with AOP

Building Software Product Line (SPL) with AOP is an approach to deal with modularization of product variabilities through aspects. The basic idea is creating the product

line core, which is common to all product line instances, using both classes and aspects. But the most important use of aspects is to introduce the variabilities (*e.g.*, alternative persistence mechanisms, performance enhancements, platform specific code) required by each product instance.

Some works present the process of creating a product line based on AOP [4, 5, 6, 26, 25] through extraction of variabilities to aspects. Also, they present strategies to migrate variabilities from conditional compilation to aspects [7].

However, we noticed that the resulting design usually contains several dependencies between core components and the components implementing the variabilities, specially when they are extracted from existing classes. With this in mind, it is easy to understand why it is difficult to make internal changes in core components without breaking aspects, demonstrating that the modularity is compromised. Certainly, SPL implemented with AOP can benefit from an approach that establishes an interface between classes and aspects, clearly defining the dependencies between them, like ours.

Chapter 3

LSD: A Language for Specifying Design Rules in AO Systems

In this chapter, we present a *Language for Specifying Design Rules (LSD)*. Some LSD concepts were sketched in previous works [70, 31] and in a more recent work [72] we presented the current form of the language, giving details about the language semantics.

As mentioned before, modularity is an important software quality attribute. Modular software accepts interventions that involve only the target module and the interface of referred modules. This means that a developer attempting to understand or modify a module should not need to modify or understand the rest of the system. The main objective of LSD is to decouple classes and aspects, improving modularity (class and crosscutting) and maximizing independent development opportunities. Through the definition of Design Rules we argue that both class and aspect developers can work independently if a minimum set of constraints is defined and respected. LSD was defined as a mechanism for expressing and checking design rules during all development phases, specially during software design.

3.1 LSD Overview

Many Object-Oriented (OO) programming languages provide the concept of interface which specifies a public set of methods and constants to be provided by any class that implements it. An interface decouples a class from its clients. Programming languages frequently support the creation of separated and narrow interfaces to different clients of the same class. Usually these interfaces contain specifications of the expected behavior of each method, through comments in natural language (like Javadoc in Java programs), aiming to hide implementation details from their clients.

The concept of interface supported by LSD, which we call *Design Rule* (DR), goes beyond the one typically supported by OO languages, involving more than public methods and constants. It can include, for example, required join points, private members, inter-type declarations and expected inheritance relationships. A DR contains a set of rules that must be followed by classes, interfaces and aspects that implement the DR. LSD is focused on specifying the *minimum requisites* about the structure and behavior of the modules that will be independently developed. This is similar to how the list

of methods in a Java interface works. Implementing classes must define the methods that appear in the interface, but are also allowed to define other methods. A DR can be seen as a constraint over the modules of a system that must hold for the system to be considered valid. So, system modules can exhibit any structure or behavior their designers see fit, unless something is explicitly prohibited by the DR specification.

To explain how design rules can be created, we describe major steps that should be followed when building an AO system with design rules.

Discuss Design Rules: Developers, based on previous experience and on a specification of the system requirements, discuss how classes and aspects will interact and agree on some design rules (Section 3.1.1).

Write Design Rules: The agreed design rules are written in some form that developers can understand. LSD supports this task by providing a new language construct that is called *dr* (Section 3.1.1). With LSD, it is possible to express these rules in a declarative fashion and verify them during implementation.

Develop OO and AO components: Each class, interface, and aspect is implemented and automatically checked against the appropriate DRs (Sections 3.1.2 and 3.1.3).

Determine the DR instance: This task involves the binding of design rules, classes, interfaces and aspects. It is necessary because classes, interfaces and aspects are not coupled and make no reference to each other. LSD determines this binding at compilation time through a Design Rule Instance (Section 3.1.4).

In the following sections, we present an example that demonstrates LSD constructs and their details. The Display Update example was chosen because it is used in several works on aspect-oriented software development. Basically it is part of a simple tool for editing drawings comprising figure elements like points and lines that are depicted in a display.

3.1.1 Discussing and Writing a Design Rule

As a result of the first step (discussing the design rules), developers could agree on the following rules for figure elements, display, and display update:

Rule 1: `FigureElement` methods called `set*` (starting with `set`, like `setX`) and `moveBy` must be `public` and return `void`. Also, all constructors must be public.

Rule 2: `FigureElement` constructors and methods called `set*` or `moveBy` are the only possible points of state change in figure elements.

Rule 3: Methods called `set*` or `moveBy` and *constructors* must change some attribute of the figure element.

Rule 4: Methods called `set*` or `moveBy` cannot call any method called `set*` or `moveBy` from a `FigureElement`.

Rule 5: A `Display` class must have a `public void update` method.

Rule 6: The aspect responsible for updating the display must declare a pointcut called `stateChange` that intercepts calls to the methods and constructors that change the state of figure elements based on their names (predetermined).

Rule 7: The display update aspect must also contain an advice that calls `Display.update` after a state change. This method cannot be called from any other place in the system.

LSD supports two kinds of rules: structural and behavioral. *Structural Rules* describe constraints about classes, interfaces, aspects and their members. Their format is similar to an interface description in Java, but it is possible to include additional constraints (beyond those traditionally required for public methods and constants), like attributes that must be declared, required private, or protected methods and expected inter-type declarations. A structural rule can also specify constraints about aspect structure, such as required pointcut and advice declarations. Listing 3.1 shows a design rule called `DisplayUpdateDR` that contains three structural rules: `FigureElement` (Lines 2–20), `Display` (Lines 22–24), and `DisplayUpdate` (Lines 26–35). These elements are *roles* that actual classes, interfaces, and aspects will perform when the system is implemented. For each role, developers must implement an actual type (or a set of types) that satisfies the role constraints. A design rule does not refer directly to actual type implementations, but defines their expected structure and behavior by means of the structural rule elements. The first line of the Listing 3.1 contains the list of roles (between brackets).

Each structural rule in a design rule specifies elements about which developers of other structural rules in the same design rule must be aware of at design and implementation time. For example, developers of the `FigureElement` class must be aware of both the `Display` class and the `DisplayUpdate` aspect. In the same way, developers of `DisplayUpdate` must be aware of the two classes in the design rule.

Behavioral Rules provide a mechanism for specifying constraints about the behavior of classes and aspects. Examples of behavioral rules are required method calls (`call/xcall`), and attribute accesses (`get/xget`) and changes (`set/xset`), as shown in Listing 3.1 (Lines 12–13, 17–18 and 33).

The first rule is expressed in Listing 3.1 within the `FigureElement` Structural Rule (Lines 3–4). Line 3 can be understood as: *"all constructors, independently of their parameter list, must be public"*. Similarly, Line 4 can be read as: *"all methods that have names starting with 'set', plus the set of methods called* ***`moveBy`****, must be* ***`public`****, return* ***`void`****, and can have any parameter"*. Any modifier or return type that appears associated with a member (`public` and `void` in the example) must match the classes implementing the `FigureElement` structural rule.

Rules 2 and 3 are expressed in Listing 3.1 (Lines 6–8) by the behavioral rule `xset`, which requires an assignment to some `FigureElement` attribute within change methods and prohibits changes elsewhere. We can read Line 6 as: *"exists some method whose name starts with 'set', contains an assignment to some attribute from FigureElement, and is the only method that performs an assignment to this attribute"*. Both Lines 7 and 8 can be read in an equivalent form. If Rule 2 were suppressed, we could have used `set` instead of `xset`, because `set` does not prohibit assignments from other places of

the program, like `xset` does. Both `set` and `xset` define the places where an assignment to an attribute must occur, but `xset` restricts the assignments to the scope in which it is present. Equivalent notion is used in the `get`/`xget` and `call`/`xcall` behavioral rules.

Listing 3.1: Design Rule for Display Update.

```
1  dr DisplayUpdateDR [FigureElement, DisplayUpdate, Display] {
2     class FigureElement {
3        all( new(..) ) then ( public new(..) );
4        all( * set*(..) + * moveBy(..) ) then ( public void *(..) );
5
6        * set*(..)    { xset(* FigureElement.*); }
7        * moveBy(..)  { xset(* FigureElement.*); }
8        new(..)       { xset(* FigureElement.*); }
9
10       all( * set*(..) + * moveBy(..) )
11       then ( * *(..) {
12                 !call(* FigureElement.set*(..));
13                 !call(* FigureElement.moveBy(..));
14               } );
15       all( new(..) )
16       then ( new(..) {
17                 !call(* FigureElement.set*(..));
18                 !call(* FigureElement.moveBy(..));
19               } );
20    }
21
22    class Display {
23       public void update();
24    }
25
26    aspect DisplayUpdate {
27       public pointcut stateChange(FigureElement fe): target(fe) &&
28          (call(* FigureElement.set*(..))    ||
29           call(* FigureElement.moveBy(..)) ||
30           call(FigureElement.new(..)));
31
32       after(FigureElement fe): stateChange(fe) {
33          xcall(* Display.update());
34       }
35    }
36 }
```

Rule 4 is expressed in Listing 3.1 (Lines 10–19) by disallowing methods that change attributes to call themselves. We used two `all` expressions, where the first one (Lines 10–14) means: *"no methods that have names starting with 'set' or called 'moveBy' can call methods with names starting with 'set' or equal to 'moveBy' from FigureElement"*. It is important to notice that LSD does not impose any order between `get`s, `set`s and `call`s within the method. It only checks if they are present in the method or constructor scope.

Also, in Listing 3.1 (Lines 22–24) the fifth rule is expressed, forcing classes that implement the `Display` structural rule to provide a `public void update` method.

Lines (27–30) and (32–34) of structural rule `DisplayUpdate` (Lines 26–35) express, respectively, rules 6 and 7. They specify that an aspect implementing `DisplayUpdate` must provide the `stateChange` pointcut and associate it to an advice that calls `Display.update`. Moreover, this advice is the only place in the scope of the design rule that calls `Display.update`.

3.1.2 Developing OO Components

Assuming that the design rules are defined, class developers can concentrate on implementing the classes while obeying these rules. They are free to change anything except what is established on the design rule. This is similar to what happens when a class implements an interface: all the methods in the interface must be implemented by the class. The main differences are that there are more types of constraints and some of them involve more than one component, like an inter-type declaration.

Listing 3.2 shows a class named `Point` that implements `DisplayUpdateDR` as `FigureElement`. This means that all constraints (structural and behavioral rules) associated with `FigureElement` must be respected by `Point`.

`DisplayUpdateDR` is also implemented by `Screen` but as `Display`. This implies that `Screen` must provide an `update` method. Based on the design rule we can detect when a constraint is violated, avoiding to affect other components that depend on these constraints. This is important because components can be developed independently.

Listing 3.2: Classes implementing the DisplayUpdateDR.

```
1  public class Point implements DisplayUpdateDR(FigureElement) {
2     protected int x, y;
3     public Point(int x, int y) {
4        this.x = x;
5        this.y = y;
6     }
7     public void setX(int x) { this.x = x; }
8     public void setY(int y) { this.y = y; }
9     public void moveBy(int x, int y) {
10       this.x = x;
11       this.y = y;
12    }
13 }
14
15 public class Screen implements DisplayUpdateDR(Display) {
16    public void update() { /* Updates Screen */ }
17 }
```

3.1.3 Developing AO Components

The `ScreenUpdate` aspect (Listing 3.3) implements the `DisplayUpdateDR` design rule as `DisplayUpdate`. In conformity with the structural rule `DisplayUpdate`, `ScreenUpdate` defines a pointcut named `stateChange` and an after advice that calls `Display.update`.

Listing 3.3: Aspects implementing the DisplayUpdateDR.

```
1  public aspect ScreenUpdate
2     implements DisplayUpdateDR(DisplayUpdate) {
3
4     private Display display;
5     public pointcut stateChange(FigureElement fe): target(fe) &&
6            (call(* FigureElement.set*(..))    ||
7             call(* FigureElement.moveBy(..))  ||
8             call(FigureElement.new(..)));
9     after(FigureElement fe): stateChange(fe) {
10       display.update();
11    }
12 }
```

It is important to notice that `ScreenUpdate` does not refer to `Point` or `Screen`, but only to the structural rules they implement. Considering that, we can understand how

it is possible to develop the aspect without knowing about the class implementations, achieving the parallel and independent development of classes and aspects discussed in Section 2.1.

3.1.4 Defining a Design Rule Instance

One important question that arises when classes and aspects are completely decoupled is where the binding between the structural rules of a design rule and their implementing elements (classes, interfaces, and aspects) will occur. In LSD, this binding is performed at design rule instantiation, which requires the list of classes, interfaces, and aspects that will play the roles of that design rule. For example, the design rule of Listing 3.1 has three parameters (Line 1): `FigureElement`, `DisplayUpdate` and `Display`. Listing 3.4 shows the `DisplayUpdateDR` instantiation by assigning a name to the instance `DispUpd` and associating class `Point` to `FigureElement`, aspect `ScreenUpdate` to `DisplayUpdate`, and class `Screen` to `Display`. It is possible to bind multiples types to a parameter. For instance, we can bind classes `Point` and `Line` to the `FigureElement` role by providing a comma-separated list of these types in the place of `Point`. This means that they must respect the set of constraints contained in `FigureElement`. Their `implements` clause is used to check if the bindings are correct, which means that each structural rule associated to the component in the design rule instance must be in its `implements` clause.

Listing 3.4: Design Rule instance.

```
1 dri DispUpd = DisplayUpdateDR(FigureElement = Point;
2                               DisplayUpdate = ScreenUpdate;
3                               Display = Screen);
```

3.2 LSD Features

In Section 3.1 we gave an overview of LSD, showing how the language is intended to be used in the development of AO Systems. In this section, we present a more detailed description of the language grammar and semantics using examples based on the Health Watcher system [90].

3.2.1 Meta-Model

Figure 3.1 shows a LSD meta-model. A Design Rule description is composed by one or more *Structural Rules* (Section 3.2.2). They are used to define structural constraints for interfaces, classes and aspects. In other words, a structural rule can define ordinary type members, like constants and method signatures (Interface Members), attributes, constructors, and methods (Class Members), pointcuts, advice, inter-type declarations (attributes, methods, and constructors), and declare declarations (Aspect Members). Additionally, *Structural Rules* support member expressions (Section 3.2.8), quantification expressions (Section 3.2.9) and behavioral rules (Section 3.2.3) within their body. *Behavioral Rules* can be defined both within the body of a structural rule and within constructors, methods, inter-type declarations, and advice bodies.

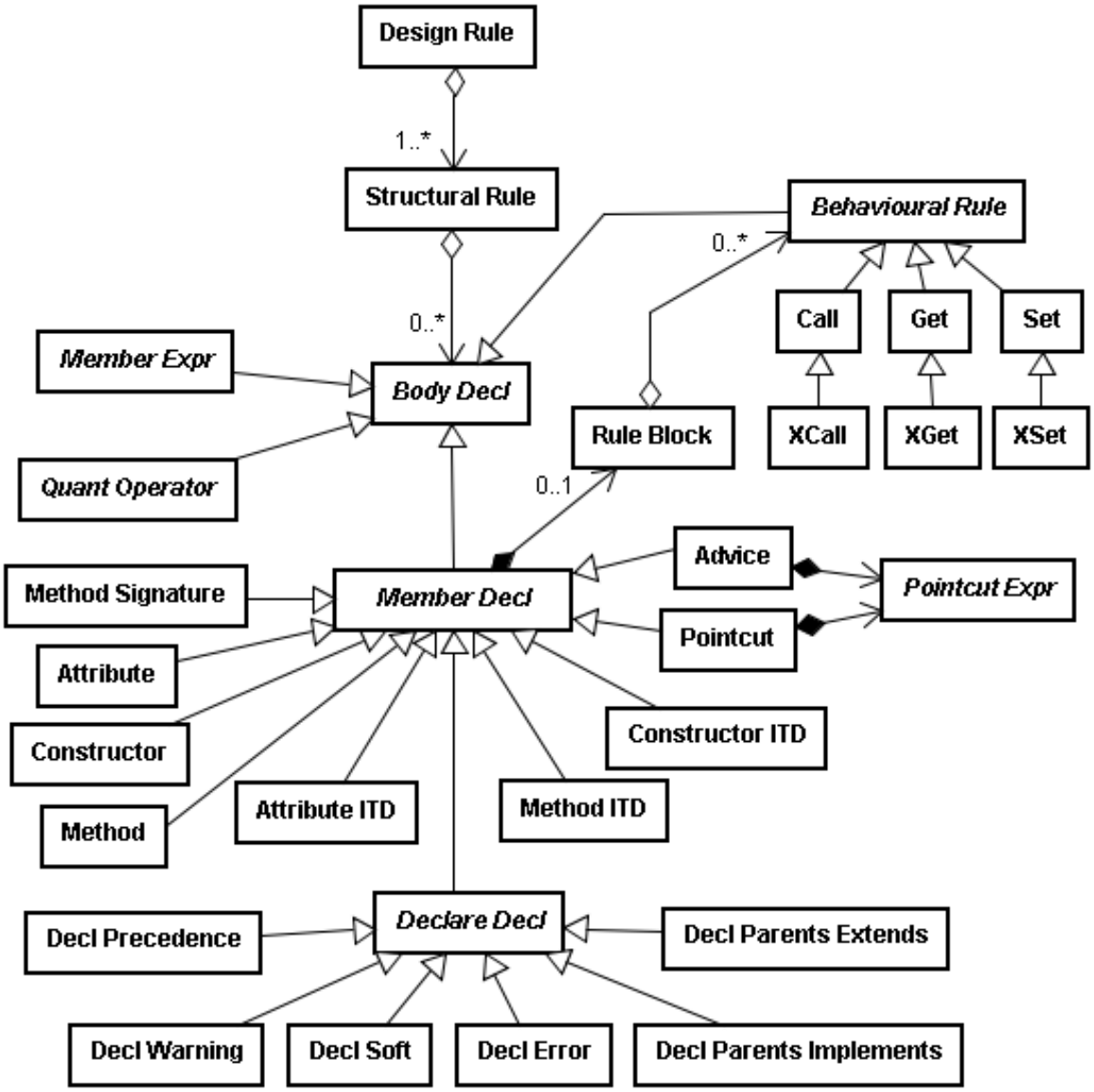


Figure 3.1: LSD Meta-Model

3.2.2 Structural Rules

Structural Rules (SR) are design rules that describe constraints over members of interfaces, classes, and aspects. They are similar to Java interfaces but they support additional constraints (beyond the traditional constraints over public methods and constants), like protected attributes that must be declared and required method calls. Moreover, there are specific constraints about aspect structure, like required inter-type, pointcut, and advice declarations.

Listing 3.5 shows part of the `RepositoryDR` rule. The latter is useful for enforcing design constraints to types of the Data Access Layer of the Health Watcher system [90]. `RepositoryDR` declares the minimum requirements so that the `Component`, `Record`, and `Repository` types can be developed in parallel.

Line 1 of Listing 3.5 contains, after the design rule name, a list of three *Design Rule Parameters*, namely `Component`, `Record`, and `Repository`. The objective of declaring those parameters is to abstract the real type names, giving support to both design rule reuse and decoupling between types. In this listing we will focus only on part of the specification of the second DR parameter (`Record`) (Lines 3–11).

Listing 3.5: Structural Rule for Repository.

```
1  dr RepositoryDR [Component, Record, Repository] {
2      ...
3      class Record {
4          public new(*);
5          public void insert(Component)
6            throws ObjectAlreadyInsertedException, ObjectNotValidException,
7                   TransactionException;
8          public Component search( (int || String) )
9            throws ObjectNotFoundException, TransactionException;
10         ...
11     }
12     ...
13 }
```

The definition of the Structural Rule `Record` (Line 3) does not impose constraints like visibility, inheritance, and interface implementation over types that implement it because no specification related to that is defined. For example, if we replace `class Record` by `public class Record`, the required type visibility becomes `public`. On the other hand, the body of Structural Rule `Record` contains three member declarations:

- The first one (Line 4) contains a *constructor declaration*, which must be `public` and have exactly one parameter, no matter the type. It is easy to identify a constructor declaration within a DR because it is always named `new` and has no return type.

- Following the constructor (Lines 5–7), there is an *instance method declaration*. The method must be called `insert`, have `public` visibility, a `void` return type, and a parameter of type `Component`. Moreover, its `throws` clause must contain exactly three exceptions: `ObjectAlreadyInsertedException`, `ObjectNotValidException` and `TransactionException`. Section 3.2.11 provides more details about exceptions.

- Another *instance method* is declared within `Record` (Lines 8–9). The method must be called `search`, have `public` visibility, return a `Component`, and contain

exactly the exceptions `ObjectNotFoundException` and `TransactionException` in its `throws` clause. Method `search` must have only one parameter, either an `int` or a `String`. This does not mean that the method must accept both types of parameters. It means that if the class provides a `search(int)` method, or a `search(String)` method, or both, it satisfies the design rule constraints.

3.2.3 Behavioral Rules

Behavioral Rules (BR) provide a mechanism for specifying constraints about the behavior of classes and aspects. They are useful, for instance, because aspect developers usually trust on some method call (or attribute access/change) at pointcut specification time. However, if the developer changes the class, removing the join point shadow (the method call), the pointcut has to be adjusted. Otherwise, the aspect will behave differently from what is expected. If some behavioral rule is defined to check the method call occurrence and it is not found, the class developer is warned, avoiding the missing join point problem.

Table 3.1 shows the *behavioral rules* provided by LSD. The scope of these rules includes classes, aspects, constructors, methods and advice. A behavioral rule defined within the scope of a class or aspect has to be respected in that scope. For example, if the call is specified within a method of a SR, the corresponding class or aspect is in conformity with the design rule if it makes a call to the method in that scope. If the `call` rule is defined within the scope of a structural rule, it means that the required method call must occur at least once in the corresponding class (or aspect) by any of its members (*e.g.*, methods, constructors and advice). Another example is to use `get` or `set` in a SR, requiring a read (or write) of the specified attribute within the required scope, Every rule initiated by "x", which means *exclusive*, guarantees that the rule will be satisfied only in the defined scope and this will not be possible in any other location among the types included in the corresponding design rule instance. It is important to notice that types not included in the design rule instance are not restricted by "x" rules. For example, the `xcall` rule guarantees that a specific method will be called exclusively within the scope in which it was defined. Calls outside of that scope are not allowed, except from types that are not bound, by the design rule instance, to any structural rule of the design rule.

These rules are also useful to guarantee, for example, that a method must not be called in a given scope, by using the negation operator (!). When a *behavioral rule* is defined within the scope of a class or aspect, it is valid for the entire class. One `call` within a class indicates that a method call to a specific method must exist in some method or constructor of that class. In fact, all possible statements are considered, including the statements from static and field initializers. However, this construct does not guarantee that a call will in fact occur at runtime. For example, if a required method call depends on a condition based on user input, the call will be executed only if the condition is satisfied.

Listing 3.6 shows three rules illustrating the use of the `xcall` behavioral rule (Lines 15, 18 and 21). These three rules indicate that methods `begin`, `commit` and `rollback` from `ITransactionMechanism` must be called exclusively within the scopes of the respective advice, prohibiting calls from any other place among the types bound to struc-

Table 3.1: Behavioral Rules provided by LSD.

Rule	Description
`call(method)`	The enclosing scope must have a *call to* ***method***.
`xcall(method)`	The enclosing scope must have a *call to* ***method***. Also, it is the only place in the design rule instance that can *call* ***method***.
`get(attribute)`	The enclosing scope must have an *access to* ***attribute***.
`xget(attribute)`	The enclosing scope must have an *access to* ***attribute***. Also, it is the only place in the design rule instance that can *access* ***attribute***.
`set(attribute)`	The enclosing scope must have an *assignment to state* ***attribute***.
`xset(attribute)`	The enclosing scope must have an *assignment to state* ***attribute***. Also, it is the only place in the design rule instance that can perform *assignments to* ***attribute***.

tural rules of `TransactionManagementDR` by the design rule instance. This is useful to guarantee that no other type can call these methods, leading to transaction management errors. For example, assuming that `TransactionManagementDR` is declared and instantiated, if the method `begin` is called from any constructor, method or advice of `TransactionManagement`, an error is reported, because calls to it can only occur within the before advice associated with the `transactionalPoints` pointcut.

Listing 3.6: Behavioral Rules for Transaction Management.

```
1  dr TransactionManagementDR
2     [ITransactionMechanism, TransactionManagement, Facade] {
3
4      interface ITransactionMechanism {
5         void begin() throws TransactionException;
6         void commit() throws TransactionException;
7         void rollback() throws TransactionException;
8      }
9
10     aspect TransactionManagement {
11
12       pointcut transactionalPoints(): call(* Facade.*(..));
13
14       before(): transactionalPoints() {
15           xcall(void ITransactionMechanism.begin());
16       }
17       after() returning: transactionalPoints() {
18           xcall(void ITransactionMechanism.commit());
19       }
20       after() throwing: transactionalPoints() {
21           xcall(void ITransactionMechanism.rollback());
22       }
23    }
24
25    class Facade {}
26 }
```

Additionally, the DR declares a `Facade` structural rule which is referred by the `transactionalPoints` pointcut and has no associated constraints. For simplicity, we are assuming that all `Facade` methods are transactional. Thus, through simple rules, it is possible to improve system modularity, documenting the interaction rules between types and checking if they are respected.

It is important to notice that a class that does not implement and is not bound to (by a design rule instance) any *structural rule* defined by `TransactionManagementDR` can call the method `begin`. This happens because the design rule checking scope is restricted to the set of types that are bound to the DR. This binding is defined by the design rule instance specification. However, since it is possible to expand the set to include any number of classes, interfaces and aspects, we can adapt the scope in accordance with developers needs.

3.2.4 Explicit Implementation of Design Rules

The structural and behavioral rules are used to create design rules, that is, contracts. We need also to specify the parts involved in those contracts. This can be done by explicitly implementing design rules. So, trying to keep similarity with the concept of Java interfaces, and following the principle of establishing an interface between classes and aspects, LSD requires that both classes and aspects explicitly implement the DRs. When a class implements a DR, all constraints contained in the DR are explicit to the developer. This means that class developers are partially oblivious about aspect implementation. Only the necessary details are exposed to the developer.

Examples of components that implement `TransactionManagementDR` are shown in Listing 3.7. In the listing, interface `IPersistenceMechanism` implements the DR, playing the role of `ITransactionMechanism`. Next, the `HWTransactionManagement` aspect implements the same design rule, but as `TransactionManagement`, whereas the `HWFacade` class plays the role of `Facade`.

Listing 3.7: Explicit implementation of the Design Rule TransactionManagementDR.

```
1  public interface IPersistenceMechanism
2     implements TransactionManagementDR(ITransactionMechanism) {
3  ...
4  }
5
6  public aspect HWTransactionManagement
7     implements TransactionManagementDR(TransactionManagement) {
8  ...
9  }
10
11 public class HWFacade
12    implements TransactionManagementDR(Facade) {
13 ...
14 }
```

For each implemented design rule, we instantiate one or more DR parameters. Based on the chosen parameters, it is possible to obtain the set of constraints imposed by the design rule. For instance, if a class developer needs to make a change, it is easier to identify which design rules must be respected, facilitating evolution tasks and preventing errors. Besides, its use enables automatic verification and separate compilation because, based on the type and its list of implemented DRs, it is possible to infer dependencies on other types (*e.g.*, a class that depends on a method inter-type declaration) and compile the type without the elements on which it depends. The use of `implements` requires a class or aspect to follow all the rules defined by the corresponding *structural rule*.

An alternative to explicit implementation of design rules in classes, aspects, and interfaces would be to provide a *separated mapping* between these types and the list

of design rules implemented by them. Also, it would be necessary to inform the corresponding structural rules implemented by each type (binding to DR parameters). We did not choose this alternative because it does not make it clear to the developer which design rules are implemented by a certain type unless all existing mappings are inspected. With an explicit indication (our approach), it is easier to discover the implemented design rules in cases where dependencies could prohibit separate understanding and compilation.

Another alternative that is similar to explicitly implementing the design rules is using *annotations* to provide the list of implemented design rules. Compared to the chosen approach (explicit implementation) it is very similar because in both of them the information of which design rules are implemented is contained in classes, interfaces and aspects. The downside of the annotation approach is requiring the distribution of a default *Annotation Type* that is referred by classes, interfaces and aspects.

3.2.5 Design Rule Instantiation

Instead of explicitly implementing a DR, when a class does not depend on declarations or behavior specified by aspects, we omit the `implements` clause in the class declaration and indicate that the class takes part in the contract only when instantiating the DR. As explained in Section 3.1.4, a design rule instance (DRI) is responsible for binding classes, interfaces, and aspects to corresponding structural rules in a DR. Through this mapping, both developers and our checking tool can infer the constraints associated with each component. So, each type associated to a SR must be in conformance with the SR constraints. For example, the design rule of Listing 3.6 has three parameters (Line 2): `ITransactionMechanism`, `TransactionManagement` and `Facade`. Listing 3.8 shows the `TransactionManagementDR` instantiation by assigning a name to the instance `TraMngDRI` and associating interface `IPersistenceMechanism` to `ITransactionMechanism`, aspect `HWTransactionManagement` to `TransactionManagement` and `HWFacade` to `Facade`.

Listing 3.8: Design Rule instance.

```
1 dri TraMngDRI = TransactionManagementDR(
2   ITransactionMechanism = IPersistenceMechanism;
3   TransactionManagement = HWTransactionManagement;
4   Facade = HWFacade
5 );
```

When a structural rule does not have dependencies that demand an `implements` in the types, it is possible to bind any type, even types that do not explicitly implement the DR. On the other hand, when their `implements` clause includes the DR, it is checked if those bindings match, which means that each structural rule associated with the type by the design rule instance must be in its `implements` clause. We chose this approach because it enables us to use existing types to instantiate DRs whenever the associated structural rules are free of inter-type declarations dependencies.

Besides allowing the binding of more than one type to one DR parameter, one can also use wildcards "*", like in AspectJ, instead of concrete type names. Besides, DR instances support the negation of type(s) using the "!" operator. For instance, we can provide *all classes from a certain package except one (or more)* as a parameter at DR

instantiation. In addition to that, it is possible to use “class”, “interface”, and “aspect” to explicitly select classes, interfaces, and aspects. We detail the DR instantiation grammar in Listing 3.9.

Listing 3.9: DR instantiation grammar.

```
1 DRInstance  ::= 'dri' Id '=' Id '(' ParamIdList ')' ';'
2 ParamIdList ::= ParamId [';' ParamId]*
3 ParamId     ::= Id '='
4 IdSet IdSet ::= IdSetExpr [',' IdSetExpr]*
5 IdSetExpr   ::= TypeNamePat | TypeGroup | '!' IdSetExpr
6 TypeGroup   ::= 'class' | 'interface' | 'aspect'
7 TypeNamePat ::= Equals to AspectJ type name pattern
```

Our approach is flexible enough to support any number of instances of the same design rule with different parameters in a single system. Also, the same type can be bound to different structural rules in different design rule instances. This approach can be useful in Software Product Lines (SPL) implemented with AOP, where design rule instantiation can be used as a configuration mechanism to support any number of product instances.

3.2.6 Modifiers

One type of constraint is related to component modifiers. It might be necessary to state that a certain class has public and not package visibility. Also, it could be necessary to prohibit developers from declaring a specific method as synchronized. These constraints usually represent dependencies between aspects and classes. For instance, if an aspect implements a concurrency mechanism but the class developer (oblivious to the aspect) declares a method as synchronized in an affected class, the aspect implementation can misbehave. With LSD, developers can state these constraints and avoid hidden implementation dependencies like that.

LSD is focused on specifying the minimum requisites about type structure and behavior. This means that declarations contained in a DR are satisfied if at least the listed modifiers are included in the types implementing the DR. So, unless they are explicitly prohibited by the DR specification, other modifiers are accepted.

Listing 3.10: Structural Rule for Repository.

```
1  dr RepositoryDR [Component, Record, Repository] {
2      ...
3      class Record {
4          ...
5          public void insert(Component)
6            throws ObjectAlreadyInsertedException, ObjectNotValidException,
7                   TransactionException;
8          ...
9      }
10     ...
11 }
```

Listing 3.10 shows part of the design rule for repositories (`RepositoryDR`) presented in Listing 3.5. Observing the structural rule `Record`, it is possible to note that no constraints about class visibility are defined in the DR. So, `Record` can be implemented by a class with any visibility — `public`, `protected`, `private` and package (when none is specified). In contrast to Java, package visibility is not assumed when no visibility

modifier is declared. In LSD, package visibility is represented by the `pack` keyword. This is also valid for other class modifiers (*e.g.*, `final` and `abstract`). In contrast, the `insert` method must be `public`. Moreover, the implemented `insert` method can also be `synchronized`, `final`, and `static`, for example. If we want to prohibit the method from being declared as `synchronized`, we just use `!synchronized` in its modifier list after the `public` modifier.

For a more complete specification of types (interfaces, classes, and aspects) and their members (attributes and methods), we provide part of the LSD grammar in Listing 3.11. `ExpClassMod` represents the expressions involving class modifiers that can be used before the `class` keyword in a DR. Similarly, `ExpMethodMod` represents the expressions for method modifiers. LSD provides the relational operators `&&`, `||` and `^^` (Line 7) for, respectively, AND, OR and XOR. It also supports the negation of a modifier using ! (NOT) and parenthesis for precedence definition. For each type of member (attribute, method, constructor, pointcut, inter-type declarations) it is possible to use the listed expressions. For instance, we could change the specification of the `insert` method of the `Record` SR from Listing 3.10 to accept the `public` or `protected` modifiers, but not `static`, as shown in Listing 3.12.

Listing 3.11: Grammar for type and member modifiers.

```
1  ExpClassMod ::= ClassMod | '!' ExpClassMod | '(' ExpClassMod ')'
2                | ExpClassMod RelOp ExpClassMod
3
4  ExpMethodMod ::= MethodMod | '!' ExpMethodMod | '(' ExpMethodMod ')'
5                 | ExpMethodMod RelOp ExpMethodMod
6
7  RelOp ::= '&&' | '||' | '^^'
8
9  InterfaceMod   ::= 'strictfp' | CommonCompMod
10 ClassMod       ::= 'final' | InterfaceMod
11 AspectMod      ::= 'privileged' | 'final' | CommonCompMod
12 AttMod         ::= 'transient' | 'volatile' | 'static' | 'final' | VisMod
13 MethodMod      ::= 'abstract' | 'final' | 'native' | 'static' |'synchronized'
14                  | 'strictfp' | VisMod
15 CommonCompMod ::= 'abstract' | 'static' | VisMod
16 VisMod         ::= 'public' | 'protected' | 'private' | 'pack'
```

Listing 3.12: Modifier Expression in a Method Declaration.

```
1  dr RepositoryDR [Component, Record, Repository] {
2     ...
3     class Record {
4        ...
5        (public || protected) && !static void insert(Component);
6           throws ObjectAlreadyInsertedException, ObjectNotValidException,
7                  TransactionException;
8        ...
9     }
10    ...
11 }
```

Additionally, it is possible to define private attributes and methods and associate constraints with them. It might sound odd to define a constraint about a private member of a class, but sometimes aspects need to refer to private members. If this dependency is defined in a design rule, it is possible to explicitly express it.

3.2.7 Method Parameters

Method parameters are important to method specifications in a DR because, in conjunction with a method name, they uniquely identify one method within a type. That is why some pointcut designators (*e.g.*, `call`, `execution` and `withincode`) in AspectJ support expressions involving parameters types and method names. Since these pointcut designators depend on method parameters, LSD supports the definition of rules over parameters, aiming to avoid join point misses caused by changes to method parameters. To illustrate, suppose that the aspect developer writes a pointcut (`pointcut insertComponent() : execution(* Record.insert(Component))`) that intercepts the execution of the `insert` method from Listing 3.12 and assumes `Component` as the type of its single parameter. If the class developer adds a parameter or changes the type of the parameter, the pointcut will not capture the method execution anymore.

LSD supports the definition of rules that expose to developers name patterns for each parameter type of methods and constructors. Listing 3.13 presents a structural rule `Record` that requires an `insert` method with a parameter of `Component` type (Line 6). If the developer changes the method parameter, as discussed above, LSD reports an error, thus avoiding the missing joint point problem. The `RepositoryDR` also uses an "*" to specify an expected `public` constructor with one parameter of any type (Line 4). Also, it requires a public method called `search` whose return type is `Component` and that accepts one parameter with `int` or `String` types (Line 8).

Listing 3.13: Method Parameter Expressions.

```
1  dr RepositoryDR [Component, Record, Repository] {
2     ...
3     class Record {
4        public new(*);
5
6        * insert(Component);
7
8        public Component search( (int || String) );
9        ...
10    }
11    ...
12 }
```

Table 3.2 shows examples of supported method (and constructors) parameter expressions using the "*" and ".." wildcards available in AspectJ pointcut designators. Also, it contains examples of the LSD support to expressions for parameter type names using the ! (NOT), `||` (OR), `&&` (AND), and `^^` (XOR) operators.

3.2.8 Member Expressions

LSD also allows the use of expressions involving structural rule members. This feature enables one, for example, to include in a structural rule a constraint that requires a `public Component search(int)` method or a `public Component search(String)` method, using the disjunction operator `||`. The class implementing this structural rule will be correct if it provides at least one of them. These constraints are shown in Listing 3.14 by the structural rule `Record` (Lines 7–11). It is important to notice that we can write the rule to the `search` method more concisely if we use an OR (`||`) expression for the parameter type, as shown in Listing 3.13. However, if some other element,

Table 3.2: Examples of parameter expressions.

Parameter Expression	Description
m()	No parameters are accepted
m(*)	One parameter of any type
m(..)	Zero or more parameters of any type
m(*, ..)	One or more parameters of any type
m(T, *, ..)	One parameter of type T, followed by one or more parameters of any type
m(.., T)	One or more parameters and the last one is of type T
m(!T)	One parameter of any type other than T
m(T1 `||` T2)	One parameter of type T1 or T2
m(T* `&&` *2)	One parameter of type with name that starts with T and ends with 2
m(T* `^^` *2)	One parameter whose type name either starts with a 'T' or ends with a '2'

beyond the type name, is also different (like modifiers, return type, and method name), the member expression becomes more flexible and easier to read. It is also possible to require a `public` constructor with one parameter (Line 4) and prohibit a `public` default constructor (Line 5). This can be useful, for instance, for pointcuts that intercept the constructor call, activating an advice that changes (or copies) the parameter value.

Listing 3.14: Structural Rule for Repository with Member Expressions.

```
1  dr RepositoryDR [Component, Record, Repository] {
2     ...
3    class Record {
4      public new(*);
5      ![public new();]
6
7      [public Component search(int)
8          throws ObjectNotFoundException, TransactionException, *;]
9      ||
10     [public Component search(String)
11         throws ObjectNotFoundException, TransactionException, *;]
12    }
13    ...
14 }
```

Requiring two or more members to exist at the same time (using the conjunction operator `&&`) is also possible and is actually the default semantics for members that appear in a structural rule. For example, the structural rule in Listing 3.14 demands that implementing classes declare a constructor with a single parameter and at least one of the `search` methods because there is an implicit `&&` operator connecting these member expressions. The combination of operators is allowed, providing more expressivity. Although this is not shown in Listing 3.14 it is possible to write expressions involving different types of elements, like methods and fields.

3.2.9 Quantification

Java interfaces implicitly specify *existential quantification* of public instance methods and constants. In previous sections we presented some LSD features that support constraints that Java interfaces cannot express. In addition, LSD provides *universal quantification*. It supports expressing rules based on several members of a structural rule. For example, suppose that we need to express that all public methods from a certain class cannot be synchronized because the concurrency management is implemented by an aspect. This rule can be expressed by selecting the desired scope (*all public methods*) and applying some rule to it (`!synchronized`). In LSD, we can express this constraint by including the quantification expression in a structural rule, as shown in Listing 3.15.

Listing 3.15: Structural Rule for Concurrence Management with Quantification.

```
1 dr ConcurrencyManagementDR [SyncType, SyncAsp] {
2     class SyncType {
3         all (public * *(..))
4         then (!synchronized * *(..));
5     }
6     aspect SyncAsp {
7         pointcut syncPoints() : execution(public * SyncType.*(..));
8     }
9 }
```

Observing the quantification expression in Listing 3.15, we can notice two method signature patterns: one immediately after the `all` keyword and another one after the `then` keyword. We refer to the first one as the *Quantification Scope* and to the second one as the *Quantification Rule*, following the general form:

`Operator` (*Scope*) `then` (*Rule*);

Where:

Operator: Specifies the amount of elements (*e.g.*, one, none, or all) from the scope that must respect the Rule. Table 3.3 contains the list of the Quantification Operators supported by our language.

Scope: Establishes the scope over which the rule is checked, selecting the members, according to the member expression, among methods, constructors, inter-type declarations (methods and attributes), pointcuts, and advice (with or without body).

Rule: Expresses the rule that must be true for the selected members. Any valid member expression can be used here.

With quantification expressions, we can write more elaborated design rules with a clear semantics. LSD supports quantification expressions with behavioral rules and also set operators "+" (Union) and "-" (Intersection). For example, we can use these features to write the rule "No method except `getInstance` can call the singleton class constructor" (shown in Listing 3.16), which is useful to write a design rule to check if a class respects the constraints of the Singleton design pattern.

Table 3.3: Quantification operators semantics.

Parameter Expression	Description
all	all selected members (N = Scope size)
exists	at least one (N > 0)
one	exactly one (N = 1)
none	zero (N = 0)
opt	at most one (N = 0 or N = 1)
range[min-max]	range from min to max (min <= N <= max)

N is the number of members from the Scope that respect the Rule

Listing 3.16: Example of Quantification Expression (Singleton).

```
1 dr Singleton [S] {
2    class S {
3       none ( * *(..) - * getInstance() )
4       then ( * *(..) { call(*.new(..)); } );
5       ...
6    }
7 }
```

Although LSD was designed aiming to support the parallel development of classes and aspects, with quantification operators it also supports the specification of some design rules related to program design. We can enforce, for instance, that *"The number of public methods in a class must range between 1 and 10"* and that ‘*'No public attributes are accepted"* through the design rule `QualityDR` from Listing 3.17.

Listing 3.17: Checking Design Quality with Quantification Expressions.

```
1 dr QualityDR [C] {
2    class C {
3       range[1..10] (* *(..)) then (public * *(..));
4       none (* *) then (public * *);
5    }
6 }
```

It is important to notice that these quantification expressions are limited to structural rule members (*e.g.*, attributes, methods and constructors). We do not allow them to express global rules to Classes, Interfaces and Aspects. However, it is possible to express quantification through DR instance bindings (shown in Section 3.2.5). The binding, which may use quantification over type names, is responsibility of a design rule instance.

3.2.10 Implements and Extends

Another type of constraint requires classes implementing a certain *structural rule* to implement or extend a specific type. We show an example of this constraint in Listing 3.18. Classes that implement the `Record` structural rule must explicitly implement the `BaseRecord` interface. The structural rule `Repository` extends class `AbstractRepository`. Both `BaseRecord` and `AbstractRepository` are existing interface and class, but could be structural rules if they were defined in the design rule `PersistenceDR`. This would improve flexibility by enabling the mapping of these design rule elements to different existing interfaces and classes. These constraints are useful for writing generic pointcuts based on the implemented interfaces and superclasses. The

design rule prevents class developers from changing `implements` and `extends` clauses and removing required join point shadows.

Listing 3.18: Implements/Extends constraints.

```
1 dr PersistenceDR [Record, Repository] {
2     class Record implements BaseRecord {
3         ...
4     }
5     class Repository extends AbstractRepository {
6         ...
7     }
8     ...
9 }
```

Both `extends` and `implements` clauses in a SR represent the *minimum requirements* for the type that implements it. In other words, the type is free to implement or extend other types beyond the defined in the SR. However, the latter case only applies to interfaces because classes and aspects do not support multiple inheritance.

Table 3.4: Implements and Extends clauses expression semantics.

Implements/Extends Expression	**Description**
`class C implements I`	C implements at least I.
`class C implements *`	C implements at least one interface.
`class C implements !I`	C implements at least one interface whose name is not I.
`class C implements I, I2`	C implements at least the interfaces I and I2.
`class C implements I* && *2`	C implements at least one interface with the name starting with `I` *and* ending with 2.
`class C implements I* || *2`	C implements at least one interface with the name starting with `I` *or* ending with 2.
`class C implements I* ^^ *2`	C implements at least one interface with a name that either starts with `I` *or* ends with 2.

Table 3.4 shows some examples of expressions involving `implements` clauses. LSD uses the same syntax to express constraints to the `extends` clause. The default semantics for the list defined in the `implements` and `extends` expression is an inclusion checking. In other words, for each name (or name expression) contained in the list, LSD searches for some type name that matches it among the type names from the implements/extends clause of the type that implements the SR.

Although this default semantics has been sufficient in most of the cases, we found some specific constraints that demand more expressivity. A simple example is trying to enforce that a class cannot implement any interface. With the *includes* semantics this is impossible. Also, we are unable to express that a class must implement a certain number of interfaces. In summary, in some cases we need a semantics that refers to one

specific element but in other situations, we need to express constraints over the complete list of elements.

Since the includes semantics covers most of the common cases, we adopted it as the default semantics, maintaining the similarity with the Java/AspectJ syntax. However, aiming to cover the other cases, we enhanced the `extends/implements` clause of structural rules, allowing the developer to choose explicitly the semantics it needs in order to express the design rule. We introduced three possibilities of explicit semantics, namely: `includes`, `excludes` and `exactly`. They are used after the `implements/extends` keyword and can be used in conjunction, except for `exactly`. Below we describe in more detail how each one of them works.

Includes: Requires that each name of the list from the SR matches at least one element from the corresponding list of types.

Excludes: Accepts any list of types which does not contain any of the names (or expressions) declared in the excludes list (from SR). An empty list of types satisfies any excludes list.

Exactly: Requires that each name of the list from the SR matches at least one element from the corresponding list of types, and vice-versa. In addition to that, the number of elements of both lists must be equal. However, there is no restriction to the ordering of the elements.

To illustrate, we provide examples of uses of `includes`, `excludes` and `exactly` in a design rule specification in Listing 3.19. The class that implements the structural rule `Record` from `PersistenceDR` must contain an `implements` clause that includes the interface `BaseRecord` and excludes all interfaces except the ones whose name ends with 'Record'. `Repository`, on the other hand, must extend the `AbstractRepository` class and no other. We provide more examples in Tables 3.5, 3.6 and 3.7.

Listing 3.19: Includes/Excludes/Exactly example.

```
1 dr PersistenceDR [Record, Repository] {
2     class Record implements includes(BaseRecord) excludes(!*Record) {
3         ...
4     }
5     class Repository extends exactly(AbstractRepository) {
6         ...
7     }
8     ...
9 }
```

This feature is also present in the `throws` clause from methods and constructors (Section 3.2.11), but with a different default semantics when compared to the `implements` and `extends` clauses. The latter two implicitly assume the `includes` semantics as default, contrasting to the `throws` clause, which assumes the `exactly` semantics. This difference is due to the fact that method and constructor callers must be prepared to catch or rethrow all the exceptions present in the `throws` clause of the called method. Hence, following the minimum requirement principle in this case would mean that method implementations could define exceptions in their `throws` clauses that might not be present in the corresponding SR. This could effectively break the clients of the method. As

Table 3.5: Exactly examples.

SR specification	Type names list	Satisfies	Explanation
`exactly(I)`	I	true	
`exactly(*)`	I	true	
`exactly(*)`	I, I2	false	Expected one element, but found two.
`exactly(*,*)`	I, I2	true	
`exactly(*,*)`	I	false	Expected two elements, but found one.
`exactly(I,I2||I3)`	I2,I3	false	Required element I not found.
`exactly(!I)`	I2	true	
`exactly(I,I2)`	I, I2	true	
`exactly(I||I2)`	I	true	
`exactly(I||I2)`	I2	true	
`exactly(I||I2)`	I, I2	false	Expected one element with name I or I2, but found two.

Table 3.6: Excludes examples.

SR specification	Type names list	Satisfies	Explanation
`excludes(I)`	I	false	An element I cannot be in the list.
`excludes(*)`	I	false	Element (I) with name matching * (all names) cannot be in the list.
`excludes(*)`	I, I2	false	Elements (I,I2) with name matching * (all names) cannot be in the list.
`excludes(*)`		true	Only matches with the empty list.
`excludes(*,*)`	I, I2	false	Elements (I,I2) with name matching * (all names) cannot be in the list.
`excludes(!I)`	I2	false	I2 satisfies the element !I from the excludes list.
`excludes(I,I2)`	I	false	I is in the excludes list.
`excludes(I||I2)`	I	false	I is in the excludes list.
`excludes(I||I2)`	I2	false	I2 is in the excludes list.
`excludes(I||I2)`	I3	true	I3 is different from both I and I2.
`excludes(I||I2)`	I3, I4	true	Neither I nor I2 appear in the list of type names.

Table 3.7: Includes examples.

SR specification	Type names list	Satisfies	Explanation
`includes(I)`	I	true	
`includes(I)`	I, I2	true	
`includes(*)`	I	true	
`includes(*)`		false	The includes list requires some element with any name.
`includes(*,*)`	I, I2	true	
`includes(!I)`	I2	true	
`includes(I||I2)`	I	true	
`includes(I||I2)`	I2	true	
`includes(I||I2)`	I3	false	No element I or I2 found in list I3.
`includes(I,I2)`	I, I2	true	
`includes(I&&I2)`	I, I2	false	There is no single element that is simultaneously named I and I2

a consequence, from the developer's point of view, the `throws` clause of a method or constructor signature in a SR has an equivalent semantics to a common method or constructor signature. However, when an aspect developer writes a pointcut based on the the `throws` clause, the `includes` semantics is more appropriate because the pointcut matches the method call if the method declares at least the exception referred to by the pointcut in its `throws` clause. LSD allows developers to choose the semantics they need using the `includes`, `excludes` and `exactly` keywords.

3.2.11 Exceptions

Aiming to support parallel development of classes and aspects, we also need to define design rules related to exceptions. They constitute dependencies between class and aspect developers that should be clearly stated. AspectJ provides forms of intercepting method executions and calls to methods that declare that they might throw exceptions. It is possible to write pointcuts to explicitly choose which method executions/calls should be captured based on their `throws` clause. This mechanism, associated with the `declare soft` declaration, enables exception handling within advice, localizing exception handling in a single unit.

LSD, besides the `declare soft` construct discussed in Section 3.2.15, allows the inclusion of the `throws` clause in methods of structural rules. The idea is to force developers to follow constraints related to exceptions that must be included or not in method declarations.

Listing 3.20 shows the design rule `TransExceptionHandling` which declares a structural rule `Component` with a method quantification expression which requires all methods called `insert` to have a single exception in their `throws` clauses,

`TransactionException`. If a class implementing the `Component` structural rule is changed and the `TransactionException` exception is removed from the `throws` clause of its `insert` method or if any other exception is added to the `throws` clause, an error is reported, avoiding problems to classes that call the `insert` method. Additionally, it declares a structural rule, `ExceptionHandler`, which exposes the dependency of the exception handler aspect on the `throws` clause. `ExceptionHandler` has a pointcut, declare soft and an around advice that depend on the `throws` clause of the `insert` method.

This example shows that establishing design rules and expressing them in a declarative way allows developers to work in parallel with more confidence. Also, it prevents some problems that could easily lead to rework when the implicit rules are not respected. With LSD, the dependencies between the components can be expressed in a separate element (design rule), exposing only the necessary details and redirecting the mutual dependencies between components to dependencies between each components and the design rule. For example, Listing 3.20 establishes that all `insert` methods must throw exactly one exception: `TransactionException`. Except for that, no other constraint is associated to `insert` methods, allowing class developers to change parameters, return type, modifiers and the statements that are included in the method body, and more importantly, without affecting the `handlingPoint` pointcut.

Listing 3.20: Design Rule with throws clause.

```
1  dr TransExceptionHandling[Component, ExceptionHandler] {
2     class Component {
3        all (* insert(..))
4        then (* *(..) throws TransactionException);
5     }
6     aspect ExceptionHandler {
7        pointcut handlingPoint() :
8           call(* Component.insert() throws TransactionException);
9
10       declare soft: TransactionException : handlingPoint();
11
12       void around() : handlingPoint() {}
13    }
14 }
```

Table 3.8 shows a list of `throws` clause examples and their respective semantics. We tried to keep the semantics of the `throws` clause as similar as possible to the semantics of the AspectJ pointcut designators `call` and `execution`. LSD enables the explicit definition of design rules that apply to the `throws` clause of methods (also ITD) and constructors, so that aspects dependent on them do not break whenever a class developer changes fragile `throws` clauses.

In an Object-Oriented program, changes to a `throws` clause usually affect callers of the respective method. For example, in Listing 3.21 if the `throws` clause of the `insert` method of class `ComplaintRecord` is changed, the following effects can be observed:

1. If method `insert` does not declare that it throws `TransactionException`, an error in class HWFacade will be reported during compilation (*unreachable catch block*). This occurs because class HWFacade calls `insert` from a `try/catch` block that is prepared to handle `TransactionException`. Since the exception will not be raised anymore, the handler becomes useless.

2. If method `insert` declares that it throws an additional exception `E`, an error in class

Table 3.8: Throws Clause Expression semantics.

Throws Expression	Description
`void m()`	No constraint about exceptions.
`void m() throws`	Method `m` does not throw exceptions (requires that there is no throws clause).
`void m() throws E`	Method `m` throws only `E` (no other exceptions are accepted, not even `E`'s subclasses).
`void m() throws E,*`	Method `m` must throw `E` (subclasses are not accepted) and one more exception.
`void m() throws !E,!E`	Method `m` must throw two exceptions, but none of them can be `E`.

`HWFacade` will also be reported during compilation (*unhandled exception type*). This occurs because the `HWFacade` class does not expect that type of exception.

Listing 3.21: Exceptions in Object-Oriented Programs.

```
1  class ComplaintRecord {
2    public void insert(Complain c) throws TransactionException {}
3    ...
4  }
5  class HWFacade {
6    public void insertComplain(Complain c) {
7       try { complaintRecord.insert(c); }
8       catch(TransactionException e) { /* Exception handling */ }
9    }
10   ...
11 }
```

Listing 3.22: Exceptions in Aspect-Oriented Programs.

```
1  class ComplaintRecord {
2     public void insert(Complain c) throws TransactionException {}
3     ...
4  }
5
6  class HWFacade {
7     public void insertComplain(Complain c) {
8        complaintRecord.insert(c);
9     }
10    ...
11 }
12
13 aspect ExceptionHandler {
14    declare soft: TransactionException :
15                  call(* *.*() throws TransactionException);
16
17    pointcut pc() : call(* *.*() throws TransactionException);
18
19    void around() : pc() {
20       try { proceed(); }
21       catch (TransactionException e) {/* Exception handling */
22    }
23    ...
24 }
```

On the other hand, in Aspect-Oriented programs, there is a different type of dependency between aspects and methods with a throws clause. In AspectJ it is possible to

write pointcuts that refer to method executions/calls based on the `throws` clause of the respective methods. Changes to the `throws` clause can create unexpected or missing join points. Considering the example in Listing 3.22, if the `throws` clause of the `insert` method from `ComplaintRecord` is changed, the following effects can be observed:

1. If `insert` method is changed so that it does not include `TransactionException` in its `throws` clause anymore:
   - No Error in class `HWFacade` will be reported because of the `declare soft` included in the `ExceptionHandler` aspect, which is responsible for handling the exception `TransactionException` that could be thrown when calling `insert` (Line 8).
   - The compiler only issues a warning if no other method throws `TransactionException` (no calls are captured).
2. If method `insert` is changed and the exception `E` is added to its `throws` clause:
   - Error in class `HWFacade` (*unhandled exception type*) because it is not prepared to handle `E`.
   - Aspect `ExceptionHandler` still captures the same join point set because `insert` continues to include `TransactionException` in its `throws` clause.

Taking in consideration the different needs of OO and AO developers, and the impossibility of choosing a single semantics that satisfies both needs, LSD assumes a default semantics to the `throws` but, similarly to the `implements` and `extends` clauses (Section 3.2.10), allows developers to explicitly select the most adequate semantics for each case. The options available are `exactly` (the default to the `throws` clause), `excludes` and `includes`. We present some examples for each of them in Tables 3.5, 3.6 and 3.7, respectively.

3.2.12 Pointcuts

Similarly to other members, pointcuts also constitute a dependency between aspects and classes. However, differently from methods or attributes, which are explicitly referred to by clients, a pointcut can refer to more than one member at a time and use partial information about it (*e.g.*, the parameters or return type of a method) and ignore its name, for example. That is why we argue that they must be declared in DRs, becoming visible to both class and aspect developers in a contract. A SR can contain pointcut declarations. In order to check if an aspect provides a correct pointcut, we require that the pointcut expression matches the pointcut expression declared in the DR. We can use different approaches to compare pointcut expressions, which we describe below:

Syntactically Equal: A pointcut expression declared in the aspect must be syntactically equal to the one defined in the DR. It is easier to check but limits the aspect developer to rewrite what is defined in the DR.

Syntactically Equivalent: The pointcut expression declared in the aspect must be syntactically equivalent to the one defined in the DR. In other words, the expression must contain the same expression but it can be written in a different form. This method is a bit more complex to check than the Syntactically Equal method, but does not bring much flexibility to the developer.

Semantically Equivalent: The pointcut expressions must capture the same set of join points. This method is more flexible but requires a complex analysis of the pointcut expressions to compare the sets of join points that they select.

Inherited: This method allows developers to use the pointcut declared in the DR as if they were defined in the aspect. This approach avoids pointcut redefinition when the pointcut expression is defined in the DR.

Listing 3.23: Design Rule with pointcut declaration.

```
1  dr TransExceptionHandling[Component, ExceptionHandler] {
2     class Component {
3        all (* insert(..))
4        then (* *(..) throws TransactionException);
5     }
6     aspect ExceptionHandler {
7        pointcut handlingPoint() :
8           call(* Component.insert() throws TransactionException);
9        ...
10    }
11 }
```

Listing 3.24: Checking a pointcut declaration.

```
1  aspect TransactionExceptionHandler
2           implements TransExceptionHandling(ExceptionHandler) {
3     ...
4     pointcut handlingPoint() :
5           call(* Component.insert() throws TransactionException);
6
7     void around() : handlingPoint() {
8        try { proceed(); }
9        catch (TransactionException e) {/* Exception handling */
10    }
11 }
```

Listing 3.25: Inheriting a pointcut declaration.

```
1  aspect TransactionExceptionHandler
2           implements TransExceptionHandling(
3           ExceptionHandler) {
4     ...
5     void around() : handlingPoint() {
6        try { proceed(); }
7        catch (TransactionException e) {/* Exception handling */
8     }
9  }
```

LSD uses the first one (*Syntactically Equal*), but also accepts the last one (*Inherited*), aiming to create more opportunities to reuse. For instance, when a DR contains a pointcut declaration (like the SR `TransExceptionHandling` in Listing 3.23), the aspect developer has two options:

Declare a syntactically equal pointcut: LSD checks for each aspect that implements a SR with a pointcut declaration, if they declare a pointcut with the same name. When the name matches, the other parts of the pointcut must also match. However, in the case of the pointcut expression, they must be syntactically equal, as we illustrate in Listing 3.24.

Inherit the pointcut defined in the DR: When no pointcut with a name that matches the pointcut name from the SR is found, LSD generates a pointcut declaration based on the pointcut declaration from the SR an adds it to those aspects that do not provide the pointcut. Listing 3.25 omits the required pointcut declaration but allows developers to use the pointcut as if it was declared within the aspect (illustrated by the *around advice*). This alternative avoids source code duplication.

Even though the inheritance approach seems to be simple — a translation from a pointcut declared in a SR to a pointcut in an aspect — we need to consider that the pointcut expression can refer to some SR, as occurs in Listing 3.23 with `Component`. The main question is how to translate references to SRs when the pointcut is inherited. We decided to use the mapping from the DR instance to exchange the references to a SR by the types associated with it. So, the reference to `Component` is exchanged by `ComplaintRecord` leading to the pointcut expression: "`call(* ComplaintRecord.insert() throws TransactionException)`".

In addition to that, we have found a more complex situation, when a SR is bound to more than one type by the design rule instance. In this case, we have adopted a different approach: building a pointcut expressing with an *OR* of the pointcut designators that make reference to a SR. For example, if `Component` is bound to `ComplaintRecord` and `EmployeeRecord`, the pointcut generated by LSD would be instead: "`(call(* ComplaintRecord.insert() throws TransactionException) || call(* EmployeeRecord.insert() throws TransactionException))`".

3.2.13 Advice

Compared to methods and constructors, advice do not have clients that depend on them because they cannot be called directly. Instead, they are activated whenever a join point matched by their pointcut is reached. Therefore, there is no apparent reason to include an advice declaration in a design rule. However, we found some cases where class developers depend on aspect behavior (calling some method, for example), as the transaction management concern implemented with aspects does. In this case, it is interesting to expose to class developers that the calls to `begin`, `commit` and `rollback` are executed by specific advice and their pointcuts. LSD checks if the aspect, which implements the SR with an advice, declares an advice that matches the advice specification of the DR. It compares the advice type (before, after or around), the parameter types, exceptions, the pointcut and the advice body statements. If some of these do not match, LSD reports an error. We show examples of advice in Listing 3.6.

3.2.14 Inter-type Declarations

Inter-type declarations (ITD) enable aspects to introduce attributes and methods in classes. These declarations are very useful, but they may also create dependencies between classes and aspects. Among other applications, this mechanism is frequently used in software product lines implemented with aspects to introduce variable method implementations or attribute values in classes [4]. Also, the template method design pattern uses inter-type declarations to introduce the template method. Design rules can be used to explicitly create a contract between classes and aspects introducing product line variabilities, as illustrated in the example of Listing 3.26. It shows a design rule, `ScreenAtributtes`, that declares two structural rules: `MainScreen` and `SizeVariability`. The last one requires two inter-type declarations of fields `WIDTH` and `HEIGHT` in the class that implements the structural rule `MainScreen`. `ScreenAtributtes` also requires from the `SizeVariability` aspect an inter-type declaration of a method called `paint` in the `MainScreen` class. As a consequence, the class developer can be sure that these two attributes and the method will exist (introduced by some aspect) and can use them within the class.

Listing 3.26: Inter-type declarations in SR.

```
1  dr ScreenAttributes [MainScreen, SizeVariability] {
2      class MainScreen {
3          ...
4      }
5      aspect SizeVariability {
6          public static final int MainScreen.WIDTH;
7          public static final int MainScreen.HEIGHT;
8          public void MainScreen.paint(Graphics);
9      }
10 }
```

The checking of inter-type declarations from a DR is relatively simple when their target is a type (not a SR). In this case, LSD searches an inter-type declaration with the same target, name and parameters (for methods). If some matching declaration is found, it also checks if the other elements match, namely modifiers, type (in the attribute case), return type, `throws` clause, and body. In summary, the inter-type declarations checking is similar to the common attribute and method declarations.

Similarly to pointcut declarations in a DR, inter-type declarations have different behaviors when they use SR names as targets of inter-type declarations. Basically we transform the inter-type declaration in the aspect that implements the DR, exchanging the reference to the target by the SR name, obtained from the design rule instance. Moreover, when the SR is associated with more than one type, we *clone* the inter-type declaration, creating one instance for each type name associated with the SR, and exchange the target name by one of the type names. After that, we have one inter-type declaration for each different target. Illustrating that, if `MainScreen` is bound to a class called `MainCanvas` and `SizeVariability` to an aspect named `DefaultScreen`, first we check if `DefaultScreen` provides the three inter-type declarations, as requires `ScreenAttributes`. Then, we replace the SR name `MainScreen` by the type name `MainCanvas` in the inter-type declarations from aspect `DefaultScreen`.

On the other hand, if `MainScreen` is bound to more than one class, copies of the inter-type declaration are created in each type to which `MainScreen` is bound. In the

case of a method inter-type declaration, the body is also duplicated.

A consequence of our approach is that it avoids both the explicit use of an artificial interface and the declaration of one declare parents for each target type, as it is usually done when we need an inter-type declaration with multiple targets. With LSD, this is hidden from developers, since they only need to provide the list of types associated to a SR in the design rule instance.

3.2.15 Declare Declarations

Another kind of inter-type declaration available is declaring that a certain type implements additional interfaces or extends a different type (that must be a subtype of its original supertype). In AspectJ, this feature is implemented by the **declare parents** construct. Although it respects the *Substitution Principle* [64], other classes and aspects can assume that the class has a new supertype because of the aspect containing the `declare parents`. This can introduce an error if the aspect is removed. LSD provides a way of expressing the `declare parents` in a DR. But, instead of checking if the `declare parents` is declared by the type that implements the DR, we introduce a corresponding `declare parents` to the type, avoiding source code redundance. For example, if a DR contains the declare parents "`declare parents: Record extends BaseRecord`", LSD exchange the SR name Record by the type associated to it and introduces the declaration "`declare parents: ComplaintRecord extends BaseRecord`" instead. When the SR is associated to more than one type, we introduce one `declare parents` for each of them. For instance, if `Record` is also associated to `EmployeeRecord`, we also introduce the declaration "`declare parents: EmployeeRecord extends BaseRecord`" into the type. Moreover, if `BaseRecord` is a SR it is exchanged by the type (or list of types) associated to it by the design rules instance.

Secondly, a `declare precedence` can be included in a DR and, at compilation time, be introduced in the type that implements the DR. But, before the introduction occurs, each reference to a SR from the `declare precedence` is exchanged by the corresponding list of types associated to them by the design rule instance. For instance, if we include "`declare precedence: TransactionManagement, ExceptionHandler`" in a DR, both `TransactionManagement` and `ExceptionHandler` are exchanged by the types associated to them by the design rule instance, in the case `HWTransactionManagement` and `TransactionExceptionHandler`, respectively. In summary, the introduced declaration corresponds to `declare precedence: HWTransactionManagement, TransactionExceptionHandler`.

Similarly, LSD provides a way of requiring that aspects implementing a DR include a **declare soft** statement. We consider this important because it exposes to the class developer what aspect is responsible for the softened exception. Also, the explicit declaration exposes the dependency with respect to aspect in a contract. A `declare soft` declaration is syntactically equals to the corresponding AspectJ declaration, but the semantics in LSD is that each `declare soft` included in a DR is translated to an AspectJ `declare soft` and introduced in the type that implements the SR. It is important to notice that the process of exchanging the references to SR names by the corresponding types occurs before the introduction. Also, the pointcut expression is translated as explained in Section 3.2.12.

Finally, LSD supports the inclusion of **declare error** and **declare warning** in a DR, allowing designers to count with the same mechanisms used by XPI developers [94, 43]. In summary, these declarations are useful to prohibit some join points to occur, raising an error or warning to developers at compilation time. A LSD advantage is that it supports the use of SRs in the pointcut expression part, giving more flexibility to developers in the same way that it creates more reuse opportunities.

3.2.16 Structural Rule type kind expression

LSD allows developers to declare a SR without explicitly associating a kind of type (class, aspect or interface) to it. This flexibility is useful, for instance, when used in conjunction with the design rule instance support to quantification over type names (Section 3.2.5), which allows the binding of type names to the DR parameters, through simple names, names with wildcards "*" and also the selection of types names based on its kind (class, aspect or interface).

Listing 3.27: Structural Rule type kind expression.

```
1  dr RemoteCallsDR [RemoteFacade, RemoteClass, CommonType] {
2      class RemoteFacade {
3          call(* RemoteClass.*(..));
4      }
5
6      class RemoteClass {}
7
8      class || aspect CommonType {
9          !call(* RemoteClass.*(..));
10     }
11 }
```

To illustrate that, we show `RemoteCallsDR` in Listing 3.27. This DR establishes that calls to any method from a `RemoteClass` must come from `RemoteFacade`. In order to express that, we put a `call` behavioral rule within the `RemoteFacade` body, which means that the `RemoteFacade` class (or classes) must contain at least one call to some method from a `RemoteClass`. We also put a `!call` behavioral rule within the `CommonType` body, prohibiting any call to `RemoteClass` methods. This SR has a type kind expression that requires that only classes and aspects are associated to it by a design rule instance. We can even omit the type kind, indicating that both classes, aspects or interfaces are allowed. We could have used the type kind expression "`!interface`" instead of "`class||aspect`" with the same result.

Listing 3.28: DR instance with type kind and wildcards.

```
1  dri remote = RemoteCallsDR(
2      RemoteFacade = HWFacade;
3      RemoteClass = *Remote;
4      CommonType = class, aspect, !*Remote, !HWFacade
5  );
```

In addition to the design rule, we need to specify the design rule instance, which we show in Listing 3.28. The `RemoteFacade` parameter is mapped to the `HWFacade` class, the `RemoteClass` parameter to all classes with name that ends with "Remote", and the `CommonType` parameter to all classes and aspects, except the ones with names that end with `Remote` (remote classes) and equal to `HWFacade` (facade class). It is important to

notice that we do not make any restriction to interfaces because they cannot contain method calls.

3.2.17 Design Rule Inheritance

With design rule inheritance, we can compose constraints. For example, observing the DR `TransactionManagementDR` from Listing 3.6 we can note that the DR depends on the adoption of the Facade design pattern [38]. Although the Facade is a feasible solution, we might choose a different mechanism to establish the set of methods that should have Transaction Management, like a naming convention to Record creation and update methods. So we might, for example, declare a `TransactionManagementDR` that focuses on specifying the expected behavior — (calls to transactional methods `begin`, `commit` and `rollback`) — at transactional points. These transactional points are matched by the `transactionalPoints` pointcut and lead to the activation of certain advice. Listings 3.29 shows a general DR for Transaction Management, while Listing 3.30 shows how to extend it to use an approach that depends on a Facade class. The approach that depends on name convention is shown in Listing 3.31. Through DR inheritance, we can identify reusable parts of the design and delay some decisions, or even provide more than one design option.

Listing 3.29: General Transaction Management DR.

```
1  dr TransactionManagementDR
2     [ITransactionMechanism, TransactionManagement] {
3
4      interface ITransactionMechanism {
5         void begin() throws TransactionException;
6         void commit() throws TransactionException;
7         void rollback() throws TransactionException;
8      }
9
10     aspect TransactionManagement {
11
12       pointcut transactionalPoints();
13
14       before(): transactionalPoints() {
15           xcall(void ITransactionMechanism.begin());
16       }
17       after() returning: transactionalPoints() {
18           xcall(void ITransactionMechanism.commit());
19       }
20       after() throwing: transactionalPoints() {
21           xcall(void ITransactionMechanism.rollback());
22       }
23    }
24 }
```

Listing 3.30: Transaction Management based on a facade.

```
1  dr TranMngFacadeDR
2     [ITransactionMechanism, TransactionManagement, Facade]
3     extends TransactionManagementDR[ITransactionMechanism,
4                                      TransactionManagement] {
5
6     aspect TransactionManagement {
7        pointcut transactionalPoints(): call(* Facade.*(..));
8     }
9     class Facade {}
10 }
```

Listing 3.31: Transaction Management based on naming conventions.

```
1  dr TranMngRecordDR
2     [Component, Record, ITransactionMechanism, TransactionManagement]
3     extends TransactionManagementDR[ITransactionMechanism,
4                                      TransactionManagement] {
5
6     class Component {}
7     class Record {
8        public void insert(Component);
9        public void update(Component);
10    }
11    aspect TransactionManagement {
12       pointcut transactionalPoints():
13          execution(public void Record.insert(Component)) ||
14          execution(public void Record.update(Component)));
15    }
16 }
```

3.3 Changes to AspectJ

Design rules are new elements to both Java and AspectJ. Aspects, after compilation and weaving are transformed in classes Java so that an AspectJ program can execute as any Java program. Therefore, no change to the Java Virtual Machine (JVM) is required. We, as well, kept compatibility with the JVM. In order to do that, we had basically two options:

1. Transforming a DR into an interface with no methods or constants just to keep some kind of reference from other types to the element that represents the interface after the compilation process.

2. Discarding the DR after checking the types that declare to implement it during the compilation process and apply some kind of renaming mechanism to change references to Structural Rules by references to the real types.

We have chosen this last option and give more details about that in the following sections. We started by making some changes to the AspectJ language in order to better support the use of design rules from classes and aspects and also to reduce the coupling between the real types implementing the design rule, enabling their independent development. One of these changes is supporting the use of Structural Rule names to instantiate types using the `new` keyword (Section 3.3.1). Another change is supporting the use of Structural Rule names everywhere a type name can be used, like variable declarations and method parameters (Section 3.3.2). Also, considering these previously mentioned changes, we introduced the control to which members of the real type implementing a structural rule are being referred by classes and aspects (Section 3.3.3).

3.3.1 Creating a Structural Rule instance within AspectJ source code

Sometimes a developer needs to create an instance (through a call to its constructor) of the real type implementing a structural rule but, although there is no restriction about

that, the class or aspect becomes dependent (coupled) to that specific real type. We provide an alternative to remove this coupling by enabling the reference to the structural rule name instead of the reference to the real type associated to it by the design rule instance.

For instance, consider an aspect that is responsible for intercepting calls to the default constructor of a database connection class (`DBCon`), and calling other constructor that receives as parameter a URL in accordance with the database in use. Listing 3.32 shows an example of this situation. Aspect `AspConfigURL` calls the Connection constructor (Line 35), but it is important to notice that the reference to `Connection`, which is a Structural Rule name (Lines 2–5), will be exchanged by a reference to the type DBCon during the compilation process, according to the mapping found at the Design Rule instance `DBConIns`. This mechanism enables the use of different `Connection` classes with the same `AspConfig` aspect and also the same `AspConfig` aspect for different `Connection` classes, mainly because they make no direct reference to each other. Without this renaming mechanism, the `AspConfig` aspect would be bound to an specific `Connection` class hindering its reuse.

Listing 3.32: Creating an instance of a Structural Rule.

```
1  dr DBConDR [Connection, AspConfig] {
2     class Connection {
3        public new();
4        public new(String);
5     }
6
7     aspect AspConfig {
8        pointcut createConnection() : call(Connection.new());
9     }
10 }
11
12 dri DBConIns = DBConDR (
13    Connection = DBCon;
14    AspConfig = AspConfigURL
15 );
16
17 public class DBCon {
18     private String url = "";
19
20     public DBCon() {}
21
22     public DBCon(String url) {
23         this.url = url;
24     }
25
26     public String getURL() {
27         return url;
28     }
29 }
30
31 public aspect AspConfigURL {
32    public static final String URL = "jdbc:jdbcvendor:data";
33
34    Connection around() : createConnection() {
35       return new Connection(URL);
36    }
37 }
```

However, this renaming mechanism has a limitation. If the structural rule `Connection` is associated to more than one real type in the Design Rule instance specification, the constructor call becomes ambiguous since it is impossible to automatically

choose between them. In this case the alternative is making a direct reference to the real type.

3.3.2 References to Structural Rules from AspectJ source code

In the same way that we can call a real type constructor using a structural rule name, it is natural to use this mechanism throughout. We have implemented this mechanism, for example to:

- Attribute type;
- Method and constructor (including inter-type declarations) return type;
- Around advice return type;
- Method and constructor (including inter-type declarations) parameters;
- Advice parameters;
- Pointcut parameters;
- Local variable declaration type;
- Exception names in throws clause;
- Exception names in catch clause;
- Declare declarations (parents, soft, warning, error and precedence).

This support would be necessary, for example, if we decided to improve the example presented in Listing 3.32 to support a connection caching mechanism, as shown in Listing 3.33. This new version requires an attribute of type `Connection` (Line 33), an around advice which returns `Connection` (Lines 35), two methods returning a `Connection` (Lines 39 and 52), a local variable declaration of type `Connection` (Line 40) and a method parameter of type `Connection` (Line 48).

In summary, the renaming mechanism is an essential feature to decouple classes and aspects because instead of making reference to real types, developers use structural rule names that are automatically exchanged by the real type names defined in the design rule instance. This feature creates opportunities to parallel and independent development of both classes and aspects in numerous situations.

3.3.3 Controlling access to Structural Rule members

Another feature related to the renaming mechanism is to control which members from the real types can be accessed through references to structural rules. In principle, the renaming enables the access to all members from real types because, during the compilation process, the references to structural rules are exchanged by the real types associated to them by the design rule instance. In order to avoid that, we created a checking mechanism that prohibit access to real type members beyond that explicitly

defined in the structural rule (as the `Connection` constructors from Listing 3.33). This approach is similar to a Java interface, but accepting more types of members, like attributes and non-public methods.

Listing 3.33: Renaming references to Structural Rules.

```
1  dr DBConDR [Connection, AspConfig] {
2     class Connection {
3        public new();
4        public new(String);
5     }
6
7     aspect AspConfig {
8        pointcut createConnection() : call(Connection.new());
9     }
10 }
11
12 dri DBConIns = DBConDR (
13    Connection = DBCon;
14    AspConfig = AspConfigURL
15 );
16
17 public class DBCon {
18     private String url = "";
19
20     public DBCon() {}
21
22     public DBCon(String url) {
23         this.url = url;
24     }
25
26     public String getURL() {
27         return url;
28     }
29 }
30
31 public aspect AspConfigURL {
32    public static final String URL = "jdbc:jdbcvendor:data";
33    private Connection cacheCon = null;
34
35    Connection around() : createConnection() {
36       return getCachedConnection();
37    }
38
39    private Connection getCachedConnection() {
40        Connection res = cacheCon;
41        if (res == null) {
42            cacheConnection( createNewConnection() );
43            res = cacheCon;
44        }
45        return res;
46    }
47
48    private void cacheConnection(Connection c) {
49        this.cacheCon = c;
50    }
51
52    private Connection createNewConnection() {
53        return new Connection(URL);
54    }
55 }
```

Chapter 4

LSD Formal Semantics

After showing LSD features in Chapter 3, in this chapter we formally define part of the language semantics. Although we do not specify the language semantics completely, we focus on the most important parts of the language.

4.1 LSD Semantics in Alloy

In this section, we present a translational semantics for our language. We map all constructions to a theory specified in Alloy [49], a formal object oriented modeling language. Alloy uses *signatures* to describe the elements presented on its model and *facts* to describe the relationship between these signatures or elements that belongs to them. We chose Alloy due to its simplicity in expressing first-order logic constraints and its tool support to perform analysis in specifications. In order to explain the translational semantics we first present our theory (Section 4.1.1), where we discuss LSD's abstract syntax encoded in Alloy, then we give an intuition of `DisplayUpdateDR` semantics (Section 4.1.2) and finally we present the translational semantics (Section 4.1.3) where we map each design rule to its counterpart in Alloy according to this theory. We also discuss the benefits and drawbacks of our approach (Section 4.1.4). More details about our theory and translations can be found in Appendix A.

4.1.1 Theory

We specifiy the abstract syntax of all elements (like classes, methods, fields, aspects, advice) in our Alloy theory (specification). For example, Listing 4.1 presents two Alloy signatures representing a class and an aspect, respectively. An Alloy signature denotes a set of objects, like a class.

A signature may introduce some relations, such as `imp`. They relate objects in one signature to another one. For instance, `imp` denotes the set of interfaces that a class may implement. The `set` keyword denotes that each object of `Class` is related to a number of objects of `Interface`. The `one` keyword denotes that each object of `Class` is related to *exactly one* element of `VisibilityQualifier`. Similarly, we have defined other relations in the `Class` signature specifying the attributes, constructors and whether the class is final and abstract. An Alloy signature may extend other signatures. In the Listing 4.1,

`Class` is a `Type`, which represents a type (class, aspect or interface). The Alloy signature `Class` is abstract. Only its subsignatures may have concrete elements. Each class is declared with a visibility qualifier (public, protected, private or package). We represent them by an Alloy signature (`VisibilityQualifier`).

Listing 4.1: Class and Aspect Representations

```
1  abstract sig Class extends Type {
2     vis: one VisibilityQualifier,
3     imp: set Interface,
4     ...
5  }
6  abstract sig Aspect extends Type {
7     attr: set Field,
8     meth: set Method,
9     advice: set Advice,
10    pcut: set PointCut,
11    decl: set InterTypeDeclaration,
12    ...
13 }
```

Following the same approach the Alloy signature representing an aspect is shown in Listing 4.1. An aspect may declare a set of attributes, methods, advice, pointcuts and inter-type declarations. Similarly, we defined an Alloy signature for each element of our language, such as attributes, constructors, interfaces, methods and advice.

4.1.2 Example

In this section, we specify the display update example using our theory. For each element presented in Listing 3.1, we create a singleton signature in Alloy. For example, Listing 4.2 declares part of the `Display` class and the `update` method (Lines 22–24 of Listing 3.1). The `one` Alloy keyword denotes that the signature contains exactly one object.

Listing 4.2: DisplayUpdateDR Semantics (Part 1)

```
1  one sig Display extends Class {}{}
2  one sig update extends Method {} {
3     vis = public
4     return = void
5     no update.param
6     update in Display.meth
7  }
```

The `Display.meth` expression denotes the set of methods declared in `Display`. Notice that there are invariants attached to `update`. It is a signature attached fact that states some constraints about `update`. For instance, it is a public void method and it is declared in the `Display` class. We add all constraints declared in all elements. The `in` keyword denotes the subset operator. It is important to observe that the `update` method cannot have parameters. To have parameters, it should have been specified as `update(..)`. It is important to observe that we do not include any constraint about the qualifiers of `update`. For example, since we do not specify whether the method is static in the declaration of `update`, this method can be static or non-static. Similarly, there is no constraint about the exceptions that the method may throw.

Listing 4.3: DisplayUpdateDR Semantics (Part 2)

```
1 one sig DisplayUpdate extends Aspect {...}{...}
2 one sig adv1 extends Advice {} {
3    adv1 in DisplayUpdate.advice
4 }
5 fact {
6    update in adv1.call
7    all m: Role - adv1 | update not in m.call
8 }
```

Listing 4.3 expresses the semantics of Listing 3.1 (Lines 32–34). It states that there is an advice declared in the `DisplayUpdate` aspect. The `all` and `not` Alloy keywords denote universal quantification and negation, respectively. Similar to a signature attached fact, an Alloy fact declares a set of invariants about the model. In the previous fact, we state that the `update` method cannot be called by any method, constructor or advice but the advice declared in the `DisplayUpdate` aspect. In our theory, the `Method`, `Constructor` and `Advice` signatures, which extend the signature `Role`, declare the relation `call`. This relation specifies all method calls that are syntactically present in the static scope of a method, constructor, or advice declaration. We map the other elements of Listing 3.1 similarly.

4.1.3 Translations

In this section we show how LSD constraints are translated into Alloy constraints. We present some of our general translations used in the example (Listing 3.1) to translate a specific design rule to its counterpart in Alloy. Translation 1 shows how the `xcall` is mapped to its counterpart in Alloy. Each translation contains two templates. The *left hand side (LHS)* template contains a *design rule in LSD*. The *right hand side (RHS)* template shows an *Alloy model* specified using our theory.

Translation 1 ⟨xcall⟩

ds
... *M*1 (...) {
xcall(... .*M*2(...));
...
}

⇒

ps
one sig *M*1 **extends** *T*{...}
{...}
one sig *M*2 **extends** *T*{...}
{...}
fact {
*M*2 **in** *M*1.*call*
all *m* : *Role* − *M*1 |
*M*2 **not in** *m.call*
}

Where:

- M1 and M2 are method identifiers.

The *ds* and *ps* meta-variables denote a set of design rule declarations and the corresponding Alloy translation, respectively. On the RHS of Translation 1 there is one fact stating that *M*2 must be called in *M*1. Additionally, it cannot be called by any other

constructor, method or advice. On the LHS, we use "..." to indicate the elements that are ignored by the transformation. These ignored elements are considered by other translations through. For instance, to `xcall`, $M1$ modifiers are not considered by this transformation. Besides, $M1$ and $M2$ are not created by this transformation. There is other transformation for method declarations that performs this task. This translation is used in the example presented in Section 4.1.2 in order to map `xcall` in Listing 4.3.

Translation 2 presents how the rules introduced by a method declaration are expressed in our language. This translation is used in the example presented in Section 4.1.2 in order to map the `update` method in Listing 4.2. We have 20 translations for classes, aspects, advice, behavior rules. Each translation deals with one construct. Therefore, Translation 2 just presents the parts of the language translated to our theory in Alloy. For example, it does not show how the parameters and qualifiers are mapped. We have other translations for them.

Translation 2 ⟨method declaration⟩

```
cds
class C {
  ... M(...){
    ...
  }
}
```

⇒

```
ps
one sig C extends Class {}
{...}
one sig M extends Method
{...} {
  M in C.meths
  ...
}
```

Parameters can be used in methods and constructors (including inter-type declarations), and have a notion of order that is considered to their identification. We defined Translation 3 to augment the Alloy model with constraints related to parameters.

Translation 3 ⟨Parameters⟩

```
... N (int) {...}
```

⇒

```
sig int_ extends Type { }
one sig N extends T {...}
{
  #param = 1
  param[0] = int_
  ...
}
```

Where:

N = name

T = method, constructor (including ITD)

When the DR uses expressions with the ".." wildcard, which means that it matches with any type and number of parameters, we use a different strategy (see Translation 4)

because the number of parameters or the exact position of a parameter in a sequence of parameters does not matter in this case.

Translation 4 ⟨Parameters⟩

... *N* (..) {...}	⇒	**sig** *int_* **extends** *Type* { } **one sig** *N* **extends** *T* {...}{}

Where:

N = name

T = method, constructor (including ITD)

4.1.4 Discussion

The first-order logic constraints can be easily mapped to Alloy, like "exists some method with name `m` in class `C`". When specifying design rules, it may happen to introduce an inconsistency, which can be detected by the Alloy Analyzer [74]. For example, consider that in the `DisplayUpdateDR` example (Listing 4.4) we have a `call` and a `xcall` to the same method `update` in different parts (`FigureElement.set*` and in the `DisplayUpdate` advice). Using our translations, we yield constraints that are contradictory (Listing 4.5).

Listing 4.4: Inconsistent Design Rule

```
1  dr DisplayUpdateDR [FigureElement, DisplayUpdate, Display] {
2     class FigureElement {
3        public void set*(..) {
4           call(* Display.update());
5        }
6     }
7     aspect DisplayUpdate {
8        public pointcut stateChange(FigureElement fe): target(fe) &&
9           (call(* FigureElement+.set*(..))  ||
10           call(* FigureElement+.moveBy(..) ||
11           call(FigureElement+.new(..)));
12
13       after(): stateChange() {
14          xcall(* Display.update());
15       }
16    }
17    ...
18 }
```

Listing 4.5: Inconsistent Alloy Constraints

```
1  update in methSet.call
2  all a: Role - adv1 | update not in a.call ...
```

As explained before, the `Role` signature represents all methods, constructors or advice. The variables `adv1` and `methSet` represent the advice declared in `DisplayUpdate` and the methods `set*` declared in `FigureElement` in Listing 4.4, respectively. This inconsistency may be difficult to detect in a larger specification. Using our approach, the Alloy Analyzer can detect that the design rule is inconsistent since we map each

design rule to a specific Alloy model. In this case, the Alloy Analyzer performs a complete analysis, since we know all elements involved. Additionally, the tool contains a functionality (the unsat core) that allows us to extract the minimum set of constraints that is making the model inconsistent. In the previous example, the Alloy Analyzer highlights the problem in the Alloy specification. As a future work, we intend to build a tool that highlights the problem in LSD.

We have a tool support that implements many of the translations proposed from LSD to Alloy (Section 4.2). We have implemented them in parallel when specifying the translation rules. The case studies and some toy examples specified in our language were mapped to their counterparts in our theory in Alloy using the tool support. Moreover, most translations are simple and deal with one construction each time. These facts increase our confidence that the set of translations is correct and covers an important portion of the language.

After the first specification of translations from LSD to Alloy, a significant set of new features were introduced in the language, requiring a revision and creation of new translations. However, our main focus is on the implementation of the static analyzer that checks the design rules of LSD, which we present in Chapter 5. The set of translations were important in the initial phases of the development of static analyzer. Below, we list features that are not currently handled by any translation, but we plan to create (or improve existent ones) as a future work.

1. Inter-type declarations of both methods and attributes;
2. Declare declarations (parents, soft, precedence, warning and error);
3. Pointcut declaration with/without expression and with/without parameters;
4. Improve advice declaration translation;
5. Expressions involving members (*e.g.*, mutually exclusive attributes);
6. Quantification operators (*exists*, *all*, *one*, *none*, *opt*, *range*);
7. Review semantics of the behavioral rules (x)call/(x)get/(x)set;
8. Support the inclusion of behavioral rules in the scope of classes, interface and aspects;
9. Usage of * and !* and expressions with !, `||`, `&&` and `^^` in:
    - Implements, Extends and Throws clauses
    - Qualifiers and Visibility Modifiers
    - Class/Interface/Aspect Members
    - Behavioral Rules
    - Member names
10. DR instance definition;

11. Association and verification of classes/interfaces/aspects listed in a DR instance with respect to DR parameters;

12. Name patterns using AspectJ wildcards in several parts of the DR declaration, like methods, attributes and parameter type names.

4.2 Translation Tool

We developed a simple translation tool as proof of concept. A portion of the LSD syntax and features are not supported yet by the tool. Given a DR, the tool verifies if there are syntactic errors and during this process create objects to keep the information extracted from each design rule. After that, these objects are analyzed, generating as output strings with corresponding specification in Alloy. The tool is very simple to use, receiving a source code (txt) containing the DR as a parameter of the Main class. As output, it produces an Alloy model that corresponds to the DR. Finally, these rules can be directly copied to the Alloy Analyzer to be executed.

We used the Java Compiler Compiler (JavaCC) [102] to implement the translation tool and currently it does not implement all LSD features. This will only be possible after concluding the revision of the existing translations to cover all LSD features (future work).

If we had a complete set of translations from LSD constraints to Alloy constraints, and also a tool to translate AspectJ source code into corresponding Alloy signatures that represent the type declarations, we could use the Alloy Analyzer [74] to check if the type declarations provided satisfy a design rule. However, the Alloy Analyzer can only clearly indicate if the types provided satisfy or not the design rule. We decided to implement a static analyzer in order to indicate more precisely the reasons why the set of classes and aspects do not satisfy the design rule.

Chapter 5

Implementation

Another contribution of our work is the development of a tool to automatically check if the design rules specified using LSD are being followed by developers. As we decided to extend the AspectJ language through the introduction of the design rule specifications (`dr`) into the language, the resulting code is not compatible with a standard AspectJ compiler. We could use some preprocessing technique to convert the source code containing classes, interfaces, aspects and design rules into pure AspectJ code (without design rules). During this preprocessing step, we could check if classes, interfaces and aspects respect the design rule constraints. However, instead of using preprocessing, we decided to extend an AspectJ compiler and try to consider design rules as similar as possible to Java/AspectJ interfaces.

Since extending the *ajc (AspectJ compiler)* [33] is a time-consuming task, because it was not designed for that, we decided to use the extensible *AspectBench Compiler (abc)* [96]. This compiler has been used to successfully implement some extensions to AspectJ [47, 44, 19, 8, 1]. Another reason to extend abc was to evaluate abc extension mechanisms. In this chapter, we present details about some tools we used, and how our tool for checking design rules was developed.

5.1 AspectBench Compiler (abc)

The AspectBench Compiler (abc) [9] is an extensible AspectJ compiler intended as a workbench for aspect-oriented language research, and it has been successfully adopted as the basis for the implementation of a number of extensions. The system is divided into a *frontend*, taking care of parsing and static semantic analysis, and a *backend*, performing optimisation and aspect weaving. The abc frontend is itself implemented as a modular extension to the *Polyglot* [73] extensible Java frontend framework. Polyglot [73] is an object-oriented framework based on standard Java which relies on extensible visitor patterns for modular extensibility [9]. Later, the abc frontend was also implemented using *JastAdd* [11], providing two alternatives to extend the abc frontend: JastAdd and Polyglot.

5.2 JastAdd

JastAdd is a metacompilation system for generating language-based tools such as compilers, source code analyzers, and language-sensitive editing support. It is based on a combination of attribute grammars and object-orientation. The key feature of JastAdd is that it allows properties of abstract syntax tree nodes to be programmed declaratively. These properties, called attributes, can be simple values like integers, composite values like sets, and reference values which point to other nodes in the abstract syntax tree (AST). The reference values allow graph properties to be defined. For example, linking identifier uses to their declaration nodes, or representing call graphs and dataflow graphs. AST nodes are objects, and the resulting data structure, including attributes, becomes an object-oriented model, rather than only a simple syntax tree [46]. In Section 5.3 we present some examples of JastAdd features that we use to implement our tool.

5.3 Extending abc to support LSD

In this section we present more details about the implementation of the *Compiler for LSD and AspectJ* (*COLA*), which is an extension to abc aiming to support the new constructs introduced by LSD. LSD has an implementation using the new frontend of abc [11], based on the aspect-oriented meta-compiler JastAdd [34]. The original frontend (that is still available) was purely object-oriented and based on Polyglot [73, 29]. We started the implementation of LSD based on this version [10, 9]. However, considering the advantages of JastAdd with respect to implementation size, better concern localization and extensions composability, we decided to rewrite the compiler.

5.3.1 Implementation Steps

We have implemented a scanner and parser to LSD source code files, without extending anything from the original abc scanner and parser. Also, we did little changes to the original abc parser. After that, we created the LSD checker, which uses both LSD and abc (AspectJ) AST nodes to, respectively, obtain the set of constraints imposed by the design rules and check if they are satisfied by the AspectJ program. Then we integrated the LSD checker with abc (see the class diagram of Figure 5.1). During the compilation process, abc activates the LSD Parser concurrently to the AspectJ parser. When both ASTs are built, abc calls the LSD checker to analyze these ASTs, recording the warnings and errors found. If no error is found, the LSD checker executes the required changes to the AspectJ AST (*e.g.*, introducing DR inherited pointcuts) before the abc backend starts the bytecode generation process. In summary, we extended the abc frontend, adding a separate AST for LSD and keeping untouched the abc backend, as shown by Figure 5.2.

We used the same tools and implementation steps adopted by other abc extensions. Since we created an independent scanner and parser, we wrote the specification for the scanner (lexical analyzer) and the parser (syntactic analyzer).

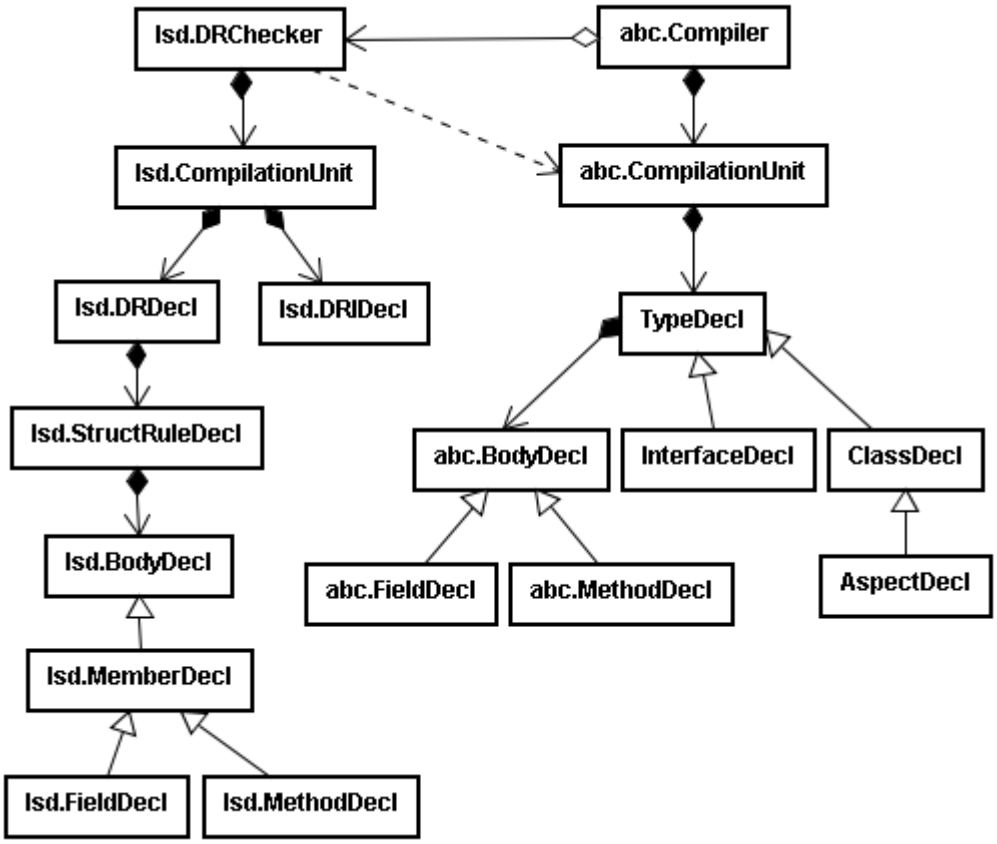


Figure 5.1: LSD and AspectJ AST

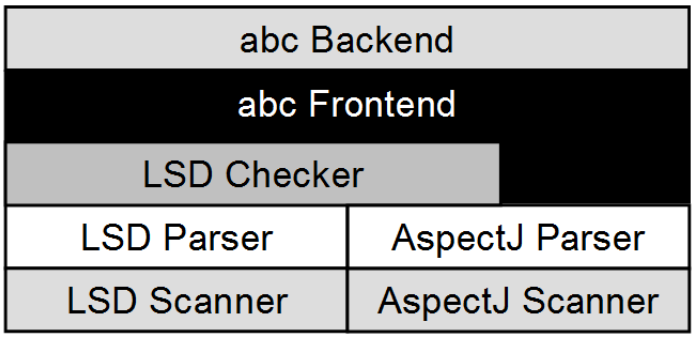


Figure 5.2: LSD and AspectJ Layers

5.3.2 Defining AST Nodes

The first step is defining the AST Nodes from LSD, which are generated by JastAdd [34] through a specification like the one presented in Listing 5.1 (Appendix B contains the complete specification of the AST nodes). This specification produces a set of classes as shown in the class diagram of Figure 5.3 and represent modifier expressions. Modifier expressions in LSD are used to impose rules about type and member modifiers (Section 3.2.6). In the following sections, we explain how we implement this feature in LSD, using the extension mechanisms provided by abc.

Listing 5.1: JastAdd AST specification example.

```
1  abstract ModifierExpr;
2
3  abstract BinaryModifierExpr : ModifierExpr ::= Lhs:ModifierExpr
4  Rhs:ModifierExpr;
5
6  OrModifierExpr : BinaryModifierExpr;
7
8  AndModifierExpr : BinaryModifierExpr;
9
10 XOrModifierExpr : BinaryModifierExpr;
11
12 NegModifierExpr : ModifierExpr ::= ModifierExpr;
13
14 BasicModifierExpr : ModifierExpr ::= Modifier;
15
16 Modifier ::= <ID:String>;
```

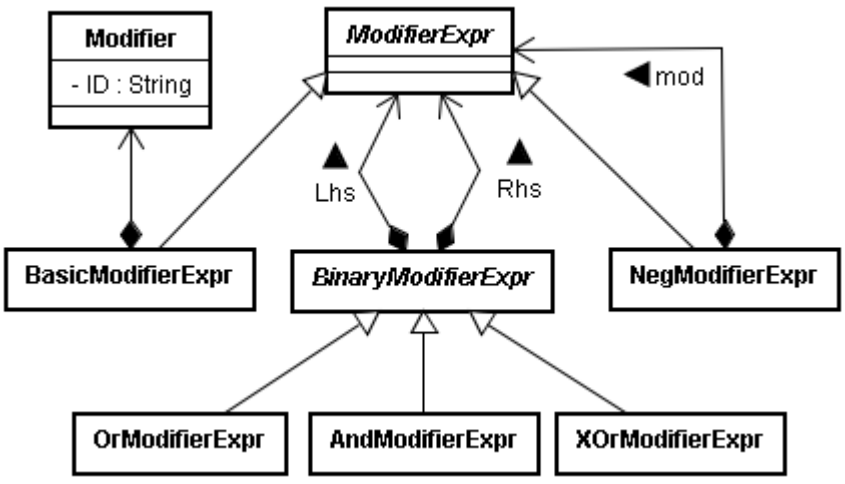


Figure 5.3: Modifier Expression hierarchy generated by JastAdd

5.3.3 Creating the Scanner

The second step was implementation of the *LSD Scanner* using JFlex [54], a lexical analyzer generator (also known as scanner generator) for Java. From the scanner specification JFlex generates a ".java" file with one class that contains code for the scanner. The generated class (LSDScanner.java) has two constructors: one that takes a `java.io.Reader` as parameter, and the other one a `java.io.InputStream` from which the input is read. The class also has a `nextToken` method that iterates through the scanner input and can be used to get the next token from the input. Both constructors and method are called by the LSD parser.

5.3.4 Creating the Parser

Next we built the *LSD Parser* using Beaver [28], which is the parser generator used by abc and its extensions. Beaver reads the LSD syntactic specification (context free grammar) and produces a Java source file (LSDParser.java) with a Java class that represents a parser for LSD.

Listing 5.2: Beaver parser specification example.

```
1  ModifierExpr modifier_expr =
2     xor_modifier_expr
3   | modifier_expr.a ORORR xor_modifier_expr.b
4         {: return new OrModifierExpr(a,b); :}
5  ;
6
7  ModifierExpr xor_modifier_expr =
8     and_modifier_expr
9   | xor_modifier_expr.a XORXOR and_modifier_expr.b
10        {: return new XOrModifierExpr(a,b); :}
11 ;
12
13 ModifierExpr and_modifier_expr =
14    unary_modifier_expr
15  | and_modifier_expr.a ANDAND unary_modifier_expr.b
16        {: return new AndModifierExpr(a,b); :}
17  | and_modifier_expr.a unary_modifier_expr.b
18        {: return new AndModifierExpr(a,b); :}
19 ;
20
21 ModifierExpr unary_modifier_expr =
22    basic_modifier_expr
23  | NOT basic_modifier_expr.a {: return new NegModifierExpr(a); :}
24 ;
25
26 ModifierExpr basic_modifier_expr =
27    modifier.a                    {: return new BasicModifierExpr(a); :}
28  | LPAREN modifier_expr.a RPAREN  {: return a; :}
29 ;
30
31 Modifier modifier =
32     PUBLIC     {: return new Modifier("public"); :}
33   | PROTECTED {: return new Modifier("protected"); :}
34   ...
35 ;
```

When the parser is executed, processing (parsing) LSD source code, it gradually creates an instance of the *Abstract Syntax Tree* (AST) corresponding to the LSD code. For example, when the parser reads a modifier expression, which can be used (among other places) in a method declaration, it creates an instance of class `ModifierExpr`

(AST node) which represents the parsed code. It is important to notice that, differently from AspectJ, LSD supports expressions involving modifiers using the *logic operators* (`||`, `&&` and `^^`).

Listing 5.2 shows a piece of the Beaver specification used to add support to *modifier expressions* to the LSD parser. For instance, the semantic action `modifier_expr` (Lines 1-5) returns an instance of the AST node `ModifierExpr` (actually an instance of its subclasses). This instance can be returned by the semantic action `xor_modifier_expr` or create by the semantic action `modifier_expr` itself. In the last case it returns an instance of the AST node `OrModifierExpr`. In the first case the semantic action `xor_modifier_expr` (Lines 7-11) is executed, returning the result of the semantic action `and_modifier_expr` (Lines 13-19) or an instance of the AST node `XOrModifierExpr`, when the XOR (`^^`) operator is found. This process is successively executed until the semantic action `modifier` is reached (terminal), returning an instance of `Modifier` (Lines 31-35) that is wrapped by a `BasicModifierExpr` created by the semantic action `basic_modifier_expr` (Lines 26-29).

5.3.5 Using the AST nodes

One of the JastAdd features is the ability to add automated support to typed traversal through the tree. For instance, the definition of `BasicModifierExpr` (see Listing 5.1) indicates that it has a nonterminal child of type `Modifier`. JastAdd automatically generates a method `getModifier` which returns the instance associated with it. Similarly, `NegModifierExpr` has a method called `getModifierExpr` that returns the `ModifierExpr` that is being negated. It is also possible to give names to children, like nodes `Lhs` and `Rhs` of type ModifierExpr, children of the `BinaryModifierExpr` node. In this case, accessor methods called `getLhs` and `getRhs` are created instead. Additionally, JastAdd includes setter methods which receive as parameter a new value to the child (*e.g.*, `setLhs` and `setRhs`). Terminal nodes are surrounded by the characters "`<`" and "`>`". In Listing 5.1, `ID`, child of the `Modifier` node, is an example of a terminal node.

Listing 5.3: JastAdd AST specification for LSD FieldDecl.

```
1  public class FieldDecl extends MemberDecl {
2      public FieldDecl(Opt<ModifierExpr> p0, IdPattern p1,
3                       List<VariableDecl> p2) {
4          ...
5      }
6
7      public boolean hasModifierExpr() {...}
8      public ModifierExpr getModifierExpr() {...}
9      public void setModifierExpr(ModifierExpr node) {...}
10
11     public IdPattern getType() {...}
12     public void setType(IdPattern node) {...}
13
14     public int getNumVariableDecl() {...}
15     public VariableDecl getVariableDecl(int i) {...}
16     public void setVariableDecl(VariableDecl node, int i) {...}
17     public void addVariableDecl(VariableDecl node) {...}
18
19     public List<VariableDecl> getVariableDecls() {...}
20     public List<VariableDecl> getVariableDeclList() {...}
21     public void setVariableDeclList(List<VariableDecl> list) {...}
22 }
```

For instance, the JastAdd definition `FieldDecl : MemberDecl ::= [ModifierExpr] Type:IdPattern VariableDecl*` produces a `FieldDecl` class that has an optional `ModifierExpr`. Also `FieldDecl` declares a mandatory `Type` which is an `IdPattern` and a set of variable declarations `VariableDecl`. The corresponding generated class (Listing 5.3) contains a constructor (Lines 2-5) and some methods, automatically generated by the JastAdd system, useful for easily accessing and changing those children nodes. It is important to notice that for list type nodes, JastAdd creates methods for accessing/changing both individual elements (Lines 14-17) and the whole list (Lines 19-21).

5.3.6 Adding features to AST nodes

The name JastAdd [34] is meant to indicate that it is easy to just add various modules to some language. These modules are written in a language based on Java. There are three main ideas that contribute to this modularity and extensibility: *object-orientation*, *static aspects*, and *declarative computations*. In addition, we can make use of imperative computations (ordinary Java code). We can extend language implementations both syntactically and computationally using these mechanisms.

JastAdd aspects support inter-type declarations for AST classes, which appear in an aspect file (see Listing 5.4). JastAdd reads aspect files and weaves the inter-type declarations into the appropriate AST classes. AST classes are the classes defined in the ".ast" files (as shown in Listing 5.1), and also the predefined classes `ASTNode`, `List`, and `Opt`.

The kinds of inter-type declarations that can occur in an aspect include ordinary Java methods and fields, and attribute grammar constructs like attributes, equations, and rewrites. Aspect files are identified by the ".jadd" or ".jrag" suffixes. Although JastAdd treats these two types of file equally, it is recommend that ".jadd" be used for declarative aspects (add attributes, equations, and rewrites) and ".jrag" for imperative aspects (add ordinary fields and methods).

`ModifierMatching` is an example of imperative aspect (Listing 5.4) created during the LSD checker implementation. It introduces the `matches` method to modifier expression AST classes. These methods have one parameter of type `abc.ja.jrag.Modifiers` that represents the AspectJ modifiers (*e.g.*, field declaration modifiers) that must match the LSD rules, expressed by the modifier expression. It is important to notice that for each `ModifierExpr` subclass (e.g. `AndModifierExpr` and `NegModifierExpr`), the aspect provides a different implementation, in accordance with its expected semantics.

JastAdd supports *attributes* in the sense of attribute grammars: attributes are declared in AST classes and their values are defined by *equations*. As in attribute grammars, an attribute is either *synthesized* or *inherited* depending on if it is used for propagating information upwards or downwards (*e.g.*, environment information for statements) in the AST. A synthesized attribute is analogous to an ordinary virtual method (where the method is a side-effect free function): The attribute declaration corresponds to the method declaration, and the equations to the method implementations. General classes can have default equations that are overridden in subclasses, analogous to method overriding.

Listing 5.4: JastAdd jrag file example.

```
1  aspect ModifierMatching {
2      public boolean ModifierExpr.matches(abc.ja.jrag.Modifiers mod) {
3          throw new RuntimeException(``This method should not be called'');
4      }
5      public boolean OrModifierExpr.matches(abc.ja.jrag.Modifiers mod) {
6          return getLhs().matches(mod) || getRhs().matches(mod);
7      }
8      public boolean XOrModifierExpr.matches(abc.ja.jrag.Modifiers mod) {
9          return (getLhs().matches(mod) || getRhs().matches(mod)) &&
10                !(getLhs().matches(mod) && getRhs().matches(mod));
11     }
12     public boolean AndModifierExpr.matches(abc.ja.jrag.Modifiers mod) {
13         return getLhs().matches(mod) && getRhs().matches(mod);
14     }
15     public boolean NegModifierExpr.matches(abc.ja.jrag.Modifiers mod) {
16         return !getModifierExpr().matches(mod);
17     }
18     public boolean BasicModifierExpr.matches(abc.ja.jrag.Modifiers mod) {
19         return this.getModifier().matches(mod);
20     }
21     public boolean Modifier.matches(abc.ja.jrag.Modifiers modifiers) {
22         return matches( generateModifierList( modifiers ) );
23     }
24     public boolean Modifier.matches(java.util.List realModifiers) {
25         if (this.getID().equals("pack")) {
26             return !realModifiers.contains("public") &&
27                    !realModifiers.contains("private") &&
28                    !realModifiers.contains("protected");
29         }
30         else {
31             return realModifiers.contains( this.getID() );
32         }
33     }
34     public java.util.List Modifier.generateModifierList(abc.ja.jrag.Modifiers m) {
35        java.util.List l = new java.util.ArrayList();
36        for (int i = 0; i < m.getNumModifier(); i++) {
37           l.add( m.getModifier(i).getID() );
38        }
39        return l;
40     }
41 }
```

For example, Listing 5.5 shows a *synthesized attribute* called `validAJDecl`. This attribute value is *true* when the node can be translated to a corresponding AspectJ node and *false* otherwise. The `FieldDecl` node (from LSD) is a valid AspectJ node (returning *true*) if its modifier expression (Lines 4-6 and 12-17), type (Line 7) and variable declarations (Line 8) are also valid.

Unlike *synthesized attributes*, *inherited attributes* were not used so frequently in the LSD checker implementation. Even so, Listing 5.6 demonstrates the use of *inherited attributes* to add to Behavioral Rule nodes access to the scope in which they were declared. This attribute is very useful to compare the scope in which the `call`/`get`/`set` is expected to occur with the scope in which it appears in the real type.

Note that the term *inherited attribute* has different meanings in attribute grammars and object-orientation. In the attribute grammar sense, an inherited attribute is an attribute whose value is defined in the parent AST node. In other words, if the `RuleBlock` node declares an inherited `getScope` attibute (Line 2), then each AST class that has a child of type `RuleBlock` must have an equation defining the `getScope` attribute of that `RuleBlock` child (Lines 3-6). This is checked by the JastAdd system which reports when some required equation is not declared.

Listing 5.5: JastAdd jadd file with Synthesized Attribute.

```
1  aspect ValidAJDecl {
2      syn boolean FieldDecl.validAJDecl() {
3          boolean res = true;
4          if (hasModifierExpr()) {
5              return res = res && getModifierExpr().validAJDecl();
6          }
7          res = res && getType().validAJDecl();
8          res = res && validVariableDecls();
9          return res;
10     }
11
12     syn boolean ModifierExpr.validAJDecl();
13     eq BasicModifierExpr.validAJDecl() = true;
14     eq AndModifierExpr.validAJDecl() = true;
15     eq OrModifierExpr.validAJDecl() = false;
16     eq XOrModifierExpr.validAJDecl() = false;
17     eq NegModifierExpr.validAJDecl() = false;
18 }
```

Listing 5.6: JastAdd jadd file with Inherited Attribute.

```
1  aspect RulesScope {
2      inh ASTNode RuleBlock.getScope();
3      eq ConstructorDecl.getRuleBlock().getScope() = this;
4      eq MethodDecl.getRuleBlock().getScope() = this;
5      eq AdviceDecl.getRuleBlock().getScope() = this;
6      eq StructRuleDecl.getBodyDecl().getScope() = this;
7
8      inh ASTNode BehavioralRuleDecl.getScope();
9      eq RuleBlock.getBehavioralRuleDecl().getScope() = getScope();
10
11     inh ASTNode RuleExpr.getScope();
12     eq BehavioralRuleDecl.getRuleExpr().getScope() = getScope();
13 }
```

The use of inherited attributes decouples an AST node from its parent because it does not need to know which parent it has (`ConstructorDecl`, `MethodDecl`, `AdviceDecl` or `StructRuleDecl`) in order to have access to the information kept by inherited attributes defined by the parent. This allows AST classes with all their behavior to be reused in many different contexts. In the Behavioral Rule case, they may occur inside many different Rule Blocks but their meaning depends only on their inherited attributes, not on any specific surrounding node.

The main reason to use synthesized attributes as a replacement to common virtual methods is that JastAdd can automatically cache the value of the synthesized attribute (in a private field), so that the method body (equation) does not have to be executed each time the attribute is accessed. This mechanism is essential to get reasonable speed when many attributes are defined. Besides, the syntax for equations is more concise than for methods, since return type and modifiers are not repeated, making aspects easier to read.

We used *synthesized* and *inherited* attributes for several purposes, usually involving numerous AST nodes. The LSD checker implementation uses these attributes to compute most of the information required to check the design rules, mainly because they are easy to define, understand and use.

5.3.7 Implementing classes to check the design rules

We created classes to navigate through both LSD and AspectJ AST nodes and check the design rules, as shown by the `check` method introduced in the `StructRuleDecl` node to check if the given `TypeDecl` matches the Structural Rule constraints (Listing 5.7). Similarly, the method `checkTypeModifiers` is introduced in the `StructRuleDecl` node and uses another introduced method called `matches`. This process was supported by synthesized and inherited attributes that were gradually incorporated into both LSD and AspectJ AST nodes. As a result, COLA produces a list of *errors* and *warnings* pointing out the design rule violations to the developer. It is important to notice that the navigation is implicitly executed by the synthesized and inherited attributes. This avoids the creation of visitors to navigate through the AST.

Listing 5.7: Introducing methods for checking the design rules.

```
1  aspect DRChecker {
2      public void StructRuleDecl.check(TypeDecl t, DRIDecl dri) {
3          checkTypeExpr(t, dri);
4          checkTypeModifiers(t, dri);
5          checkExtends(t, dri);
6          checkImplements(t, dri);
7          checkTypeBody(t, dri);
8      }
9      public void StructRuleDecl.checkTypeModifiers(TypeDecl t, DRIDecl dri) {
10         if (this.hasModifierExpr() &&
11             !this.getModifierExpr().matches( t.getModifiers() ) ) {
12             error( new LSDError(this, t, this.getModifierExpr().toString(),
13                                 t.getModifiers().toString(),
14                                 LSDError.TYPE_MODIFIERS_INCORRECT) );
15         }
16     }
17     ...
18 }
```

In our opinion, this new abc version, which provides the frontend based on JastAdd, is better than the previous one (frontend based on Polyglot), from the standpoint of the AspectJ extension developer, for a couple of reasons. First it is easier to create and reuse attributes associated to AST nodes due to the attribute grammar support. Second, these attributes can be modularized in separate files, making them easier to maintain. Finally, we do not need to create visitors to traverse the AST, collecting the required information, because we can delegate this task to the automatic attribute value computation of the JastAdd system, which includes an attribute value cache mechanisms that improves performance.

5.4 Using the LSD Compiler (COLA)

As explained before, COLA was implemented as an extension to *abc*, so we decided to maintain its invocation as similar as possible to the original *abc*. As a matter of fact, if no design rule is defined, the compiler works exactly as before.

In order to differentiate between AspectJ and LSD source code, we established that Design Rules and Design Rule instances must be defined in files with the extensions ".dr" and ".dri", respectively. Whenever file names with these extensions are provided as parameters to *COLA*, the design rule checker is executed during the compilation process, warning and pointing out errors found in the source code.

For example, when we execute the command

```
abc -ext abc.lsd *.java *.aj *.dr *.dri
```

All AspectJ (".java" and ".aj") and LSD (".dr" and ".dri") source files are processed by COLA, executing the design rule checker and generating common Java bytecode files (".class"). After compilation, the generated bytecode files can be executed by any standard Java Virtual Machine without any extra library. It is important to notice that the current version of COLA does not check class files, requiring the corresponding source code files from all types referred by a design rule instance.

Another example, discussed in Listing 1.1, shows a class `C` and an aspect `A` with a pointcut (`callToM2`) that depends on a method call to `m2` within `m1`. This dependency can be expressed in LSD by the design rule `DREx` and its instance `DRIEx` (Listing 5.8). The resulting class `C` and aspect `A` are shown in Listing 5.9.

Listing 5.8: DR and DRI used by COLA.

```
1  dr DREx [C,A] {
2    class C {
3      void m1() {
4        call(* C.m2());
5      }
6      void m2();
7    }
8    public aspect A {
9      pointcut callToM2() : call(* C.m2()) && withincode(* C.m1());
10   }
11 }
12
13
14 dri DRIEx = DREx(C = C;
15                  A = A);
```

Listing 5.9: Aspect dependent on a method call.

```
1  public class C {
2     public void m1() {
3        m2();
4     }
5     public void m2() {...}
6  }
7
8  public aspect A {
9     after() : callToM2(){...}
10 }
```

If the class developer extracts the call to method `m2` from the body of method `m1` to the body of `m3`, violating the design rule `DREx`, COLA reports an error at compilation time (Listing 5.10), giving information to the class developer about the constraints that were violated.

Listing 5.10: Error Reported by COLA.

```
1  [Error in class C] Method declaration with required behavior not
2  found:
3
4  void m1() { call(* C.m2()); }
5
6  (Check structural rule C within design rule DREx)
7
8  Found 1 error(s)!
```

Chapter 6

Evaluation

This chapter contains an evaluation of LSD. We discuss advantages and disadvantages of using our approach when compared to Crosscutting Programming Interfaces (XPIs) [94, 43]. Such interfaces specify the exposed join points by classes and aspects. Although just an abstract XPI representation was provided in by Sullivan *et al.* [94], Griswold *et al.* [43] presents how to implement some crosscutting interfaces as syntactic constructs of AspectJ. Basically, design rules are documented using *abstract pointcut descriptions* and constraints applied to classes and aspects. These constraints might be written as `declare warning` (or `declare error`) constructs or just as comments in the source code (when a constraint cannot be expressed as a `declare warning` or `declare error`).

It is important to note that the advantages of using a Design Rule based approach over an oblivious approach are discussed by Sullivan *et al.* [94]. That is why we focus on the direct comparison between LSD and the XPI approach.

6.1 Health Watcher

Our evaluation compares three specifications of the transaction concern from the Health Watcher system [90]: the `TransactionManagementDR` shown in Listing 6.3 and two XPIs introduced in the next section. We also compare the design rule specification of the distribution concern in LSD with another that uses the XPI approach (Section 6.1.2). Our goal is to discover to what extent LSD satisfies the objective of being more expressive (supporting the automatic checking of design rules) and concise when compared to XPIs. That is why the following criteria are considered [35]:

- **Language expressiveness**: quantifies the degree to which a language is able to express a constraint. In fact, it is a three level factor — language supports, does not support, or partially supports a specific rule.
- **Language conciseness**: measures how simple it is to express a constraint in a language. Here, we use the minimum number of tokens required to express a constraint as a measurement of conciseness.

6.1.1 Transaction Concern

In order to enable the parallel development between classes and the aspects responsible for implementing the transaction concern, we have to specify the following constraints:

(C1) There must exist an interface (`ITransactionMechanism`) that defines methods for starting (begin), committing (commit), and rolling back transactions (rollback);

(C2) These methods may throw an exception `TransactionException` in case of error;

(C3) Also, in order to enable the definition of the pointcuts, there should be a facade named HWFacade;

(C4) The transaction aspect `HWTransactionAspect` must call `ITransactionMechanism` methods. Moreover, these calls have to occur at specific events, detailed in what follows:

- A transaction must be started before any facade method [1] (the aspect should call `ITransactionMechanism.begin`);
- After the return of any facade method, the current transaction should be committed (the aspect should call `ITransactionMechanism.commit`);
- If any exception is raised by facade methods, the current transaction should be rolled back (the aspect should call `ITransactionMechanism.rollback`); and

(C5) These calls must only occur within the aspect `HWTransactionAspect`.

Listing 6.1 shows a first XPI specification for the transaction management concern. It is supposed to guarantee constraints C1, C3, and C5. Lines 4 – 7 specifies C1, stating that the `ITransactionMechanism` interface should exist and have `begin`, `commit` and `rollback` methods. Although we expect `ITransactionMechanism` to be an interface, we cannot enforce such a restriction using XPIs (unless in the form of comments). The *ajc compiler* [33] assumes that `ITransactionMechanism` is an interface, a class or even an aspect, reporting an error only if this type does not exist. No error is reported if that interface exists but does not define any of the mentioned methods. As we can see on lines 8 – 15, constraint C2 is described only in natural language, so it is not possible to check it in an automatic way.

The pointcut defined in Line 2 states that a class (or interface) named `HWFacade` must exist. Moreover, all methods of `HWFacade` should have a transactional context. Such a pointcut encompasses constraint C3.

[1] Although capturing calls to all facade methods is a feasible solution, but not so frequently used in practice, we might choose a different mechanism to establish the set of methods that should have transaction management, like a naming convention to transactional methods or using an additional interface that explicitly exposes the set of transactional methods. This last alternative is used in Section 6.1.2 to expose the set of remote methods but can be easily applied to the transaction management concern case.

The `staticMethodScope` pointcut defined in Line 17 states that the `HWTransactionAspect` type must exist. As aforementioned, we cannot guarantee that this type will be an aspect using XPIs. This pointcut is used on the `TransactionContractXPI` (Lines 30 – 35) to guarantee that any call to methods of the `ITransactionMechanism` interface occurs *only* within the `HWTransactionAspect`. Otherwise, an error is raised (Constraint C5).

Listing 6.1: Base version of the TransactionManagementXPI.

```
1  public abstract aspect TransactionManagementXPI {
2   pointcut transactionalMethods(): execution(* HWFacade.*(..));
3
4   pointcut callsToTransactionContext() :
5     call(void ITransactionMechanism+.begin())  ||
6     call(void ITransactionMechanism+.commit()) ||
7     call(void ITransactionMechanism+.rollback());
8   /*
9    * Methods begin(), commit() and rollback() of ITransactionMechanism
10   * can throw TransactionException in case of error. Their signatures are:
11   *
12   * void begin() throws TransactionException;
13   * void commit() throws TransactionException;
14   * void rollback() throws TransactionException;
15   */
16
17  public pointcut staticMethodScope(): within(HWTransactionAspect);
18
19  /*
20   * HWTransactionAspect must call the methods begin(), commit(),
21   * and rollback() defined in the ITransactionMechanism interface.
22   * These calls should occur within advice like the following ones:
23   *
24   * before() : transactionalMethods() {... tm.begin(); ...}
25   * after returning() : transactionalMethods() {... tm.commit(); ...}
26   * after throwing() : transactionalMethods() {... tm.rollback(); ...}
27   */
28 }
29
30 public aspect TransactionContractXPI {
31    declare error:
32       TransactionManagementXPI.callsToTransactionContext() &&
33       !TransactionManagementXPI.staticMethodScope()
34           : "Illegal use of a transactional method";
35 }
```

Finally, Lines 20 – 26 require the `HWTransactionAspect` aspect to call methods `begin`, `commit`, and `rollback` (`tm` represents an instance of one class that implements `ITransactionMechanism`). Moreover, these calls have to occur within specific advice (C4). Nevertheless, these constraints were specified using natural language, which means that no compilation error is reported if these expected calls do not occur within specific `HWTransactionAspect` advice.

We could improve this XPI to check part of Constraint C4 at compile time. In order to do that, we should introduce lines 3 – 19 of Listing 6.2 into the `TransactionManagementXPI`. This is necessary because the *ajc compiler* only warns that a given pointcut does not match *join points shadows* if the pointcut is used by some advice. That is why we define three empty advice (Lines 15 – 19). In that way, a compilation warning is raised if any of the given advice are not applied, i.e., if methods `begin`, `commit` or `rollback` of `ITransactionMechanism` are not called inside `HWTransactionAspect`. Note that `after` or `around` could have been used as advice, instead of `before`. However, despite requiring calls within the advice of

`HWTransactionAspect`, the (extended) version of `TransactionManagentXPI` does not enforce the exact advice where each call is supposed to occur. For example, although we expect a call to the `commit` method within an `after returning` advice, no error is reported if such a call happens within another advice, method or constructor. This problem arises because, since advice in AspectJ are anonymous, we cannot write pointcuts that advise a specific advice.

Listing 6.2: Extended version of TransactionManagementXPI. This version specifies that HWTransactionAspect must call begin, commit and rollback methods.

```
1  public abstract aspect XPITransaction {
2   ...
3   public pointcut expectedCallToBegin() :
4     within(HWTransactionAspect) &&
5     call(void ITransactionMechanism+.begin());
6
7   public pointcut expectedCallToCommit() :
8     within(HWTransactionAspect) &&
9     call(void ITransactionMechanism+.commit());
10
11  public pointcut expectedCallToRollback() :
12    within(HWTransactionAspect) &&
13    call(void ITransactionMechanism+.rollback());
14
15  before(): expectedCallToBegin() { }
16
17  before(): expectedCallToCommit() { }
18
19  before(): expectedCallToRollback() { }
20 }
```

Listing 6.3: Behavioral Rules for Transaction Management.

```
1  dr TransactionManagementDR
2     [ITransactionMechanism, TransactionManagement, Facade] {
3
4      interface ITransactionMechanism {
5         void begin() throws TransactionException;
6         void commit() throws TransactionException;
7         void rollback() throws TransactionException;
8      }
9
10     aspect TransactionManagement {
11
12       pointcut transactionalPoints(): call(* Facade.*(..));
13
14       before(): transactionalPoints() {
15           xcall(void ITransactionMechanism.begin());
16       }
17       after() returning: transactionalPoints() {
18           xcall(void ITransactionMechanism.commit());
19       }
20       after() throwing: transactionalPoints() {
21           xcall(void ITransactionMechanism.rollback());
22       }
23    }
24
25    class Facade {}
26 }
```

The use of LSD can mitigate these problems. The `TransactionManagementDR` (Listing 6.3) restricts that `ITransactionMechanism` must be an interface and that `TransactionManagement` must be an aspect, otherwise a compilation error is raised. In addition, the `xcall` construct of LSD assures a call occurs only within a specified

scope among the DR components. With the use of this constructor, the same DR restricts that `begin`, `commit` and `rollback` can only be called within before, after returning and after throwing advice, respectively. As a result, all presented constraints are satisfied by the LSD specification. But, as we show in the following sections, some constraints cannot be expressed by our language.

Table 6.1 summarizes our evaluation. Regarding expressiveness, both XPIs statically check C1, C3, and C5. Both C2 and C4 cannot be checked using XPIs, since they are described in natural language expressed by commented code. The pointcuts introduced in Listing 6.2 allow the partial checking of Constraint C4. However, by using these pointcuts, the *ajc* compiler only reports constraint violations if the `TransactionAspect` does not call any of the transactional methods (`begin`, `commit`, or `rollback`). On the other hand, by using LSD we could specify and statically check all the mentioned constraints.

Table 6.1: Comparison between LSD and XPI (Transaction Management).

	Expressiveness					**Conciseness**				
	C1	C2	C3	C4	C5	C1	C2	C3	C4	C5
XPI	SC	N	SC	N	SC	39	-	16	-	25
Extended XPI	SC	N	SC	P	SC	39	-	16	96	25
DR	SC		SC	SC		25		4	75	

Where the acronyms mean Statically Checked (**SC**), Partially Checked (**P**) and No Checking (**N** of natural language representation)

Considering the conciseness criterion, we could specify C4 and C5 by means of the `xcall` behavioral rule in LSD, stating: (a) calls to transactional methods must occur within `TransactionAspect` and (b) these calls must occur at a specific advice. In fact, this leads to increased conciseness, since LSD design rule requires 75 tokens (instead of 96 tokens in the extended XPI) to specify these constraints. In addition, our DR can check C4, which is not completely possible with XPIs. Moreover, constraints C1 and C2 can be easily described and statically checked with the description of the `ITransactionMechanism` interface present in Listing 6.3.

6.1.2 Distribution Concern

Another important Health Watcher concern implemented with aspects is *distribution*. In summary, class developers from the view layer use a facade to access the *business* layer as a local class. But distribution aspects intercept calls to this facade and redirects these calls to the corresponding methods from a remote facade, which, in its turn, invokes the respective remote methods. This process is transparent to local facade clients. We identify some design rules that must be established between the local facade and the distribution classes and aspects. These rules are expressed below through a set of constraints:

(C1) There must exist an interface that contains the set of local methods that must

be intercepted by the distribution aspect. In the HW case, this interface is called `IFacade` and calls to `IFacade` methods constitute the join points.

(C2) Also, there must be an interface (`IRemoteFacade`) with the same set of methods defined by `IFacade`, but with the difference that each of them contains an additional Exception (`RemoteException`) in its `throws` clause.

(C3) There must exist a class that directly implements the local facade (`IFacade`). In the case of the HW, this class is `HealthWatcherFacade`.

(C4) The remote facade class (`RemoteFacade`) must provide a static method called `getInstance`, which returns an instance of the class.

(C5) The remote facade class (`RemoteFacade`) cannot have a `main(String[])` method;

(C6) An aspect executing in the client side captures all calls to the local facade (`IFacade`), through a pointcut (`facadeCalls`), and substitutes the original call by a remote call, delegating this task to the method `MethodExecutor.invoke`.

(C7) Some aspect must declare that business classes implement the `Serializable` interface, allowing their instances to be used as parameters of remote calls;

(C8) Also, an aspect must declare that the remote facade class implements *IRemoteFacade*, aiming to enforce that all remote methods are provided by that class.

(C9) A distribution aspect must introduce a `main` method in the class that implements the remote facade `IFacade`. In the case of the HW, this class is called `HealthWatcherFacade`.

(C10) The execution of the `main` method, introduced by the distribution aspect, must activate an advice around that: (1) initializes the remote facade instance; (2) binds the *remote object* to a name in the naming service.

We show in Listing 6.4 how these constraints can be expressed in LSD through the `DistributionDR` declaration. The first constraint (C1), expressed by the SR `ILocalFacade` requires that the local facade interface to declare at least one method. C2 is only partially met because, although we can enforce that all methods from the remote facade class (`IRemoteFacade`) must declare that might throw `RemoteException`, we cannot compare this set with the set of methods defined in `ILocalFacade`. Presently, LSD does not support constraints that involve members from different SRs, but we plan to provide some mechanism to support that as future work.

Following that, we observe that C3 is completely satisfied by the SR `LocalFacade` which implements `ILocalFacade`. In the same way, `RemoteFacade` satisfies the Constraint C4, requiring the declaration of a `public synchronized static getInstance` method which returns a `RemoteFacade` (Line 13). Following that, the Constraint C5 is enforced by the SR `RemoteFacade` (Line 14), which forbids the declaration of any `main` method with a parameter of type array of `String` within the remote class. As required by the Constraint C6, `ClientDistribution` represents an aspect in the client side that declares a `facadeCalls` pointcut which captures calls to all methods from the local

facade, and also declares an around advice using that pointcut. The advice contains a `call` behavioral rule to the `MethodExecutor.invoke` method, which requires some call to this method within the advice scope.

Listing 6.4: Distribution design rules with LSD.

```
1  dr DistributionDR [Component, ILocalFacade, LocalFacade,
2                     IRemoteFacade, RemoteFacade,
3                     ClientDistribution, ServerDistribution] {
4     interface ILocalFacade {
5        exists (* *(..)) then (* *(..));
6     }
7     interface IRemoteFacade {
8        all (* *(..)) then (* *(..) throws includes(RemoteException));
9     }
10
11    class LocalFacade implements ILocalFacade {}
12    class RemoteFacade {
13       public synchronized static RemoteFacade getInstance();
14       ![* main(String[]);]
15    }
16
17    class Component {}
18
19    aspect ClientDistribution {
20       pointcut facadeCalls() : call(* ILocalFacade.*(..));
21
22       Object around() : facadeCalls()  {
23          call(Object MethodExecutor.invoke(..));
24       }
25    }
26
27    aspect ServerDistribution {
28       declare parents: Component implements Serializable;
29       declare parents: RemoteFacade implements IRemoteFacade;
30
31       public static void RemoteFacade.main(String[]);
32
33       protected Remote initFacadeInstance() {
34          call(* RemoteFacade.getInstance());
35       }
36
37       protected pointcut facadeMain(String[] arr):
38          execution(static void RemoteFacade.main(String[])) && args(arr);
39
40       void around(String[] arr): facadeMain(arr) {
41          xcall( * ServerDistribution.initFacadeInstance() );
42          call( * UnicastRemoteObject.exportObject(*) );
43          call( * Naming.rebind(..) );
44       }
45    }
46 }
```

The remaining Constraints (C7 to C10) are satisfied by `ServerDistribution`, a SR that represents the server-side aspect. It contains two `declare parents` (Lines 28 and 29) which satisfy, respectively, C7 and C8. It is important to observe that `Component` can be bound only to classes. `Component` can be associated to any set of classes through a design rule instance. Following, `ServerDistribution` declares an introduction of a `public static void main` method to RemoteFacade, satisfying the Constraint C9. Finally, respecting the Constraint C10, `ServerDistribution` declares a pointcut (`facadeMain`) that captures the execution of the introduced method (`main`), and activates an around advice that creates an instance of the remote facade and registers it in the naming service.

Aiming to compare the design rule for the distribution concern specified in LSD with a corresponding XPI, we created two aspects that represent the distribution XPI, which we show in Listing 6.5. The first one is responsible for the client-side (`ClientDistributionXPI`) and the other for the server-side (`ServerDistributionXPI`) constraints. We summarize the comparison results in Table 6.2.

Listing 6.5: Distribution design rules with XPI.

```
1  abstract aspect ClientDistributionXPI<ILocalFacade> {
2     pointcut facadeCalls() : call(* ILocalFacade.*(..));
3
4     Object around() : facadeCalls()  {
5        return doRemoteCall();
6     }
7
8     protected abstract Object doRemoteCall();
9  }
10
11 abstract aspect ServerDistributionXPI<Component, RemoteFacade, IRemoteFacade> {
12    declare parents: Component implements Serializable;
13    declare parents: RemoteFacade implements IRemoteFacade;
14
15    protected pointcut facadeMain(String[] arr):
16       execution(static void RemoteFacade.main(String[])) && args(arr);
17
18    void around(String[] arr): facadeMain(arr) {
19       createFacadeAndRegister();
20    }
21
22    protected abstract void createFacadeAndRegister();
23 }
```

`ClientDistributionXPI` declares the pointcut `facadeCalls` that intercepts calls to all methods from the type `ILocalFacade`. However, we cannot express that the calls are supposed to be redirected to a remote object, using the MethodExecutor.invoke method. So, C6 is partially satisfied by the XPI. It is important to observe that we used *generics* to decouple the XPI from the real local facade type. Since the XPI requires a type parameter, we consider that the XPI also satisfies C1. This was necessary because we assume that the developer puts all local methods in the local facade, indicating to the aspect developer the set of methods that can have the local call exchanged by a remote call. Using this approach, the class developer can establish the subset of the facade methods that can be intercepted by the distribution aspect. This approach can also be applied to the transaction management concern implementation.

After that, `ServerDistributionXPI` declares two `declare parents`, the first one declares that business classes (Component parameter type) must implement `Serializable`, satisfying C7 and requiring the same number of tokens (7) because the declarations are equal. The second declares that the `RemoteFacade` parameter type implements `IRemoteFacade` parameter type, as required by C8. However, the XPI cannot express that the aspect developer must introduce the main method, as required by C9. Finally, the advice that intercepts the main method is defined. It calls an abstract method, that must be implemented by the aspect developer, and is responsible by the remote facade initialization and registration. But, there is no way to automatically enforce this requirement with XPIs. So, the Constraint C10 is partially satisfied, and that is why it requires less tokens (64 against 92 of LSD) to be expressed.

Table 6.2: Comparison between LSD and XPI (Distribution).

	Expressiveness									
	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
XPI	SC	N	N	N	N	P	SC	SC	N	P
DR	SC	P	SC	SC	SC	SC	SC	SC	SC	SC
	Conciseness									
	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10
XPI	8	-	-	-	-	37	7	7	-	64
DR	21	26	6	23		41	11	7	12	92

Following, both C2, C3, C4 and C5 cannot be checked through XPIs. Firstly, we cannot express that all methods from the remote facade must declare that might throw `RemoteException` and check the one-to-one mapping from each local facade method to a remote facade method (both required by C2). Secondly, we cannot assert that the local facade is implemented by some class, as requires C3. Thirdly, XPIs cannot enforce that a certain class declares a *static* method, that is why it does not satisfy C4. If `getInstance` was an *instance* method, we could have used an interface to enforce its declaration. Finally, C5 is not satisfied because XPIs cannot forbid method declarations. However, we could forbid calls and executions of this method, but that would not work either. This occurs because the design rule C9 specifies that the method will be introduced by some aspect. This seems contradictory, but the idea is hindering class developer from implementing a method that will be introduced by some aspect. Listing 6.6 shows how the aspects that represent the distribution XPI are extended and integrated to the classes and interfaces from the Health Watcher system.

Listing 6.6: Defining concrete aspects for the Distribution XPI.

```
1  interface IFacade {}
2
3  interface IRemoteFacade {}
4
5  class HealthWatcherFacade {}
6
7  interface Component {}
8
9  aspect ClientDistribution extends ClientDistributionXPI<IFacade> {
10    protected Object doRemoteCall() {
11       /* Calls MethodExecutor.invoke to redirect the local call to
12          a remote call */
13    }
14 }
15
16 aspect ServerDistribution extends ServerDistributionXPI
17                <Component, HealthWatcherFacade, IRemoteFacade> {
18    protected void createFacadeAndRegister() {
19       /* Creates the facade instance and registers it */
20    }
21 }
```

Although we used just two concerns in our evaluation, we consider them representative enough in the Health Watcher context. For example, the *Repository* design rule (Listings 3.5 and 3.10) cannot be statically checked using XPIs, since the majority of the

constraints would be represented as natural language. In its turn, the *Persistence* design rule, not discussed in this work, requires the same XPIs constructs as the transaction management one. The *Exception Handling* concern can also be specified through pointcut declarations, `declare soft` declarations and methods `throws` clauses, all supported by LSD.

6.2 Design Quality Checking

We show in Section 3.2.9 that LSD supports the specification of constraints to check the design quality. For instance, in Listing 3.17 we present a design rule that checks if the number of public methods satisfies a certain range and if a class does not declare public attributes. In summary, we have two constraints:

(C1) The number of public methods must range from 1 to 10;

(C2) No public attribute is allowed.

Those kind of constraints can be expressed in LSD because we can create rules related to types structure and use quantification over their members. With XPIs we cannot write both C1 and C2 constraints because the checking mechanism used by XPIs is based on the use of `declare error/warning`, which are limited to prohibit the join point shadows captured by their pointcut expression. Attribute *declarations* (among other kinds of member declarations) do not constitute join point shadows, differently from their use (get and set).

Listing 6.7: XPI that partially enforces non public attributes constraint.

```
1 public aspect NonPublicAttributesXPI {
2     declare error : get(public * *.*) || set(public * *.*) :
3         "Public attributes are prohibited";
4 }
```

However, in the case of public attributes, we can use an alternative mechanism, like creating a `declare error` which captures any get or set of public attributes (Listing 6.7). In this case, however, we are prohibiting the *use* of public attributes and not their *declaration*, as required by C2, which means that we can create a class with public attributes without problems, violating the constraint, until we try to use those attributes from other classes, leading to an error at compilation time.

We compare both approaches and present the results in Table 6.3. Basically, LSD can express both C1 and C2, requiring for that, respectively, 23 and 12 tokens. On the other hand, the XPI approach cannot express C1. With respect to C2, we consider that it can partially check the constraint, requiring 28 tokens for that. When compared to LSD, again we observe that the specification is more concise.

Although we do not compare our language directly with other approaches besides XPIs, in the case of Listing 6.7, we expect to achieve better results — specially with respect to expressivity — with the tools we present in Section 2.3.6, namely: *Semmle-Code*, *Design Wizard* and *PDL*. This occurs because LSD was not designed aiming to express these kind of constraints.

Table 6.3: Comparison between LSD and XPI(Design Quality Checking).

	Expressiveness		**Conciseness**	
	C1	C2	C1	C2
XPI	N	P	-	28
DR	SC	SC	23	12

6.3 Discussion

Observing the comparison results from previous sections, we have some evidence that LSD enhances the XPI approach because it provides a language with a defined semantics, and that can be used to specify and automatically check more design rules than the XPI approach, which uses exclusively AspectJ to express and check the constraints. Additionally, it is important to notice that the use of LSD does not hinder the use of XPIs. As LSD is an extension of AspectJ, we can create the XPIs contracts, using `declare error/warning` or not, and also define the aspects with pointcuts that serve as interfaces between class and aspect developers. That is why in our evaluation LSD was capable of expressing more constraints than the XPI approach.

Chapter 7

Conclusion

In this work we present LSD, a language for specifying design rules in systems implemented with AspectJ. LSD assists developers to declare these design rules using an unambiguous language. We argue that through the establishment of design rules it is possible to obtain crosscutting modularity and still preserve class modularity. However, besides defining design rules, it is important to have a language (with a defined semantics) that supports expressing and automatically checking them against the code. The language semantics was partially specified in Alloy and was fine-tuned with the assistance of the Alloy Analyzer [74] and its support to automatic checking. We also implemented an extension to the AspectBench Compiler (abc) [96] to support LSD constructs. LSD expressivity was compared to the XPI approach [43] using as case study Health Watcher aspects, such as the transaction management aspect.

We discuss how LSD, with its broader interface notion, can improve modularity and serve as a solution to eliminate ambiguity and reduce the complexity found in other approaches, like XPIs implemented with AspectJ [43]. LSD allows to independently develop classes, interfaces and aspects, as long as the design rules are preestablished. We cataloged some existing design rules from the Health Watcher system and tried to express them both in LSD and XPIs, and concluded that LSD can express more constraints than XPIs, in a more concise way, and without compromising the automatic checking capability.

Another interesting point was using Alloy to specify LSD semantics. The Alloy Analyzer [74] helped us to deeply understand some constraints that we wanted to express and lead us to think about some options and take decisions that influenced LSD expressivity and understandability. This happened, for example, with the behavioral rule `xcall`. We noticed that whenever we declare two `xcall's` to the same method in different places, we are creating an inconsistent DR. Based on that, we decided to limit its scope to the components involved in the DR so that we can use two `xcall's` to the same method in different DRs.

LSD requires the creation of new artifacts (design rules) that demand enough experience from software designers. Additionally, developers must get used to new language constructs. In spite of that, explicitly expressing design rules, and specially being capable of checking them, eases the task of developing new classes and aspects and also adapting existing ones.

Design rules are important for many types of aspects, but, for general aspects that

make no reference to classes or interfaces (no coupling), like the basic Tracing and Logging aspects, we have little benefit from the use of design rules. LSD does not impose restrictions that constrain the development of these basic aspects, since it is not obligatory that aspects implement design rules. We assume that developers are capable of determining when it is worth to establish the design rules, in other words, if the design is mature enough to pay the price of writing a design rule.

Another important point is that although we consider defining design rules as a necessary step to modular AO development, it is possible to write classes and aspects, and later establish the design rules that were used. In this case, they will not help during development but will be useful for preventing errors and to assist developers during software maintenance and evolution, like Aspect-Aware Interfaces [53] (Section 7.1.3). This approach can be used whenever the best design is unknown (lack of experience) or in agile development processes, like Extreme Programming [14].

7.1 Related Work

Modularity issues in aspect-oriented programming have been reported by different authors [94, 22, 91]. For instance, Clifton and Leavens argue that existing *aspect-oriented* programming languages do not contribute to comprehensibility, since they require systems to be studied in their entirety. In order to improve comprehensibility of AOP systems, they propose a simple set of restrictions that minimizes this problem [22]. In their proposed approach, aspects are categorized as *observers* — aspects that do not change the effective specifications of the modules, or as *assistants* — aspects that might change the modules specifications. Observers preserve modular reasoning even in the cases that advised modules do not make explicit references to them. On the other hand, Clifton and Leavens argue that, in order to preserve modular reasoning, modules that are advised by assistants should make explicit references to them. A new construct (`accept type`) was proposed to indicate that a module (class or interface) accepts to be advised by an assistant aspect. This construct aim at indicating to the class developer that an *assistant* aspect exists, and should be studied when some maintenance task needs to be executed. In spite of that, it maintains the feature obliviousness with respect to the *observer* aspects. LSD also indicates to class developer the aspects that should be studied, through the Design Rules specification. These Design Rules contain information about aspects like inter-type declarations, pointcuts, advice and required/prohibited behavior. In doing so, LSD supports the *information hiding* principle, keeping the class developer away from the complete implementation of the aspect.

Other authors focus on discussing how to expose more stable aspect-oriented interfaces and how to compute module interfaces in AO systems. Since these later works are more related to our proposal, we discuss them in more detail in the remaining of this section.

7.1.1 Open Modules

Open Modules is an approach for dealing with modularity issues in AOP [2]. Besides exposing data structures and functions, Open Modules interfaces can also expose point-

cuts denoting internal semantic of events. Clients of these modules are able to advise only exported pointcuts, introducing a form of encapsulating join points occurring inside a module and protecting them from external advising. Moreover, by exporting a pointcut, the module developer is compromised to maintain the semantics of that pointcut, which, in fact, mitigates the *fragile pointcut* problem. Although join point hiding is an important concern, it does not provide information to the aspect developer (beyond exported join points and public methods), like required/prohibited inter-type declarations, method calls, attribute accesses and assignments, that are useful for decoupling classes and aspects, and consequently their developers. Differently from LSD, Open Modules does not offer any mechanisms for describing aspect developers responsibilities and the relationship between the types that participate of a collaboration. As a consequence, module developers are still able to unintentionally implement part of a concern assigned to a different team because they cannot assume the existence of any behavior expected to be modularized as an aspect.

7.1.2 Crosscuting Programming Interfaces (XPIs)

Sullivan *et al.* [94] presented a comparative analysis between an AO system developed following the widely cited oblivious approach and the same system developed with clear design rules that document interfaces between classes and aspects. This last approach promises benefits when relevant crosscutting behavior are anticipated and when new code, anticipated or not, can be written against existing interfaces (design rules). Its main problem is expressing the design rules in natural language, leading to sometimes verbose and ambiguous interpretation. With LSD, we can express most part of these design rules in an unambiguous language and also check if they are being respected, as discussed in Chapter 6.

In a subsequent work [43], they propose to express and check the design rules using AspectJ. They called this approach Crosscutting Programming Interfaces (XPI) (more details and examples in Chapter 6). XPIs authors argue that the main advantages of their approach is that: (a) it does not require any new construct in the AspectJ language; and (b) there is no restriction to the pointcut visibility. In fact, this characteristic encourage the use of XPIs in sites that are already using the AspectJ language. However, most constraints required for defining the responsibilities of both class developers and aspect developers cannot be checked using the proposed XPI language (a deeper comparison of our approach against XPIs is presented in Chapter 6). Consequently, it is not possible to guarantee, at least automatically, if certain design rules are being obeyed, like LSD does. Although it is possible to check part of the design rules using XPIs, the use of a language not designed to this purpose leads frequently to complex specifications (contracts imposed by aspects).

7.1.3 Aspect-Aware Interfaces (AAI)

Kiczales and Mezini [53] defend that the complete interface of a module can only be determined once the complete configuration of the systems is known. They introduce the notion of aspect-aware interfaces, which describe the existing dependencies between classes and aspects, giving support to reasoning about the effect of aspects over classes

by pointing out where aspects are affecting a certain class. This interface must be automatically recomputed whenever classes or aspects change and, in fact, tools like AJDT already offer similar functionality through IDE resources that indicate which advice apply in a certain point. Also, these AAIs can be saved and compared with newer ones to detect changes to the set of join points intercepted by aspects. This approach helps to identify missing and accidental join points [88]. Although very useful for evolution, this kind of interface does not support earlier software development phases, when it is necessary to partition the system into classes and aspects, and distribute tasks among development teams. At that moment, it is necessary to establish design rules that will govern both class and aspects development. LSD serves exactly to this purpose and brings automatic checking of the developed code.

7.2 Future Work

We intend to extend LSD through the addition of constructs, such as invariants, pre- and postconditions that can only be checked dynamically, probably following a similar approach to the used in JML [60, 80, 82, 83, 81]. Although, an important difference is that the constraints would be written within DRs and not in the classes. This allows to anticipate to the developer the constraints that must be respected. Also, developers of other related components can count on these constraints before the real components exist.

As another future work, we aim at building a tool for checking DR consistency, *i.e.*, if it is possible to write a program that satisfies the DR. This is useful because the developer responsible for the DR can define contradictory constraints that cannot be matched by any program. The Alloy Analyzer can be useful for implementing this tool, since it performs a complete analysis and searches by some instance. If no instance can be found, the DR is inconsistent. In addition, the Alloy Analyzer contains a functionality (the unsat core) that allows us to extract the minimum set of constraints that is making the model inconsistent, helping to identify which constraints are contradictory.

Also, we plan to perform a revision of the existent translations form LSD to Alloy to cover all LSD features. Then, based on these translations, we can implement a new translation tool from LSD to Alloy, but using abc [96]. This alternative allows the reuse of the existent parser for LSD (from COLA).

Another future work is better evaluating LSD. First, we plan to build two versions of a completely new system using LSD and XPIs and compare them considering several perspectives, including reuse. Secondly, we plan to compare LSD with other approaches, including Semmle Code [98], PDL [68] and Design Wizard [16], aiming to identify other kinds of design rules and verify if LSD can check them. As a result we can extend LSD to support more design rules.

Besides helping to specify DRs in a declarative manner and having a tool support to verify if the DRs are being respected, we intend to implement an IDE (probably Eclipse [32]) extension to support design rule creation and checking. This could be very useful for pointing to the developer the exact constraint that is not being respected. Also, this integration could include a visualization tool to, based on a design rule instance, give a view of the system configuration.

Finally, DRs written in LSD could be used to generate classes, interfaces and aspects based on their Design Rule specifications. One future work is implementing a tool integrated to Eclipse [32]. This feature is commonly found in UML [15] modeling tools [45] supporting the generation of components based on models.

Appendix A

Theory and Translations from LSD to Alloy

In this section, we give a detailed description of the translational semantics for our language. First we map all LSD constructions to a theory (Section A.1) specified in Alloy [49]. Then we present some translations (Section A.2) that support the mapping of each design rule to its counterpart in Alloy according to the theory.

A.1 Theory

This section explains the theory of our Alloy model. The theory is fundamental to understand the translation rules between DRs and our Alloy model presented in Section A.2. Our theory model is composed by signatures, each one corresponding to a component found in a DR. These signatures have the required fields (relations) to identify classes, interfaces, aspects and their elements (*e.g.*, methods and attributes).

We decided to use abstract signatures to represent components. In the same way, fields present in each component are defined as abstract signatures. In its turn, fields use logic qualifiers and new signatures in their definition. For instance, a class contain several attributes, then in the abstract signature of a class we defined a field that corresponds to these attributes and through the logic qualifier **set** we define that this field is a set of signatures of a specific type. As attributes have unique and particular characteristics, then we have a proper abstract signature that contains new fields (relations) or qualifiers. From the composition of various signatures of our model we have as result a module that contains the required theory to describe the mapping of a DR to an Alloy model.

A.1.1 Class Signature

Initially it is defined an abstract signature *Type* that is common to several signatures of our model. Then the class existence requirement can be translated to a an abstract signature with the name of the *Class*. This signature extends the generic signature *Type* which contains the required fields for reading and understanding a class in Alloy. Listing A.1 presents the abstract signature of a class in Alloy.

Listing A.1: Class Abstract Signature

```
1 abstract sig Class extends Type {...}
```

Class Signature Components

A Java class has methods, attributes, constructors, etc. Therefore, our corresponding abstract signature to this class in Alloy must reflect the existence of these components. In doing so, we use new signatures and logic qualifiers to describe them. Listing A.2 presents a new signature with the description of some of these components.

Listing A.2: Class Signature Components

```
1 abstract sig Class extends Type {
2   meth: set Method,
3   attr: set Field,
4   ...
5 }
```

Since our class signature has a set (**set**) of methods and attributes, we need to specify abstract signatures that represent these fields. In the same way that we defined a signature for a class in Listing A.1, we present signatures to methods and attributes that are used in the definition of the class signature, as shown in Listing A.2.

A.1.2 Method signature

Listing A.3 shows an abstract signature in Alloy that corresponds to a method.

Listing A.3: Abstract Method Signature

```
abstract sig Method extends Role {...}
```

In the method signature shown in Listing A.3 we define the set of fields associated to it. For example, a method has a visibility, return and parameters. Then our signature must reflect the presence of those components. This way, we show in Listing A.4 some of the components present in this signature.

Listing A.4: Abstract Method Components

```
1 abstract sig Method extends Role {
2   return: lone Type,
3   param: seq Type,
4   vis: VisibilityQualifier,
5   ...
6 }
```

We show some of the components present in a method signature and also some new logic qualifiers. A method must have only one return type, in this way we defined the return type with an unary set through the qualifier **lone**. A method must also have a sequence of parameters, where the order is important (differently from a **set**), so the parameter qualifier is **seq**. Finally, method visibility assumes predefined types which are represented by the VisibilityQualifier signature.

A.1.3 Visibility Qualifiers

Considering that the visibility of methods, attributes, classes, among others, assume predefined types (public, protected, private and package), we have several signatures associated to them, each one associated the one of these types. These signatures extend the generic signature *VisibilityQualifier*, so they have in common the fact of being visibility qualifiers, but at the same time have unique signatures(**one**). Listing A.5 presents the possible types of visibility and the abstract signature they extended.

Listing A.5: Visibility Qualifiers

```
1 abstract sig Qualifier {}
2 abstract sig VisibilityQualifier extends Qualifier {}
3 one sig public, private_, protected, friendly extends VisibilityQualifier {}
```

A.1.4 Adding Components

In the same way that we added the method signature (*meth*) to the class signature, we can add other signatures, like a visibility (*vis*), if a class is abstract (*abs*) or final (*leaf*), which is the set of interfaces that it implements (*imp*), from which class it inherits (*ext*), the set of constructors (*cons*), and so on and so forth. Listing A.6 presents a class abstract signature with all its components.

Listing A.6: Abstract Class Signature

```
1  abstract sig Class extends Type {
2    vis:  VisibilityQualifier,
3    abs:  AbstractQualifier,
4    leaf: ConstantQualifier,
5    ext:  lone Class,
6    imp:  set Interface,
7    attr: set Field,
8    cons: set Constructor,
9    meth: set Method
10 }
```

We presented in a few steps the construction of the class signature to the module of theory in Alloy. This module is composed by various others signatures, as we describe in the Table A.1.

Table A.1: Signatures from our Alloy theory.

Signature	**Corresponds to**
Class	Class in a DR
Interface	Interface in a DR
Aspect	Aspect in a DR
Exception	Exception in a DR
Method	Method in a DR
Constructor	Constructor in a DR
Field	Field in a DR
Advice	Advice in a DR
PointCut	Pointcut in a DR
InterTypeDeclaration	Inter-type Declaration in a DR

A.1.5 Theory Module

Based on the approach used in the creation of an abstract signature that corresponds to a class, we can create many other signatures, according to the requirements of our model. The class diagram shown in Figure A.1 presents all signatures from our model, including their components that define relations with other signatures.

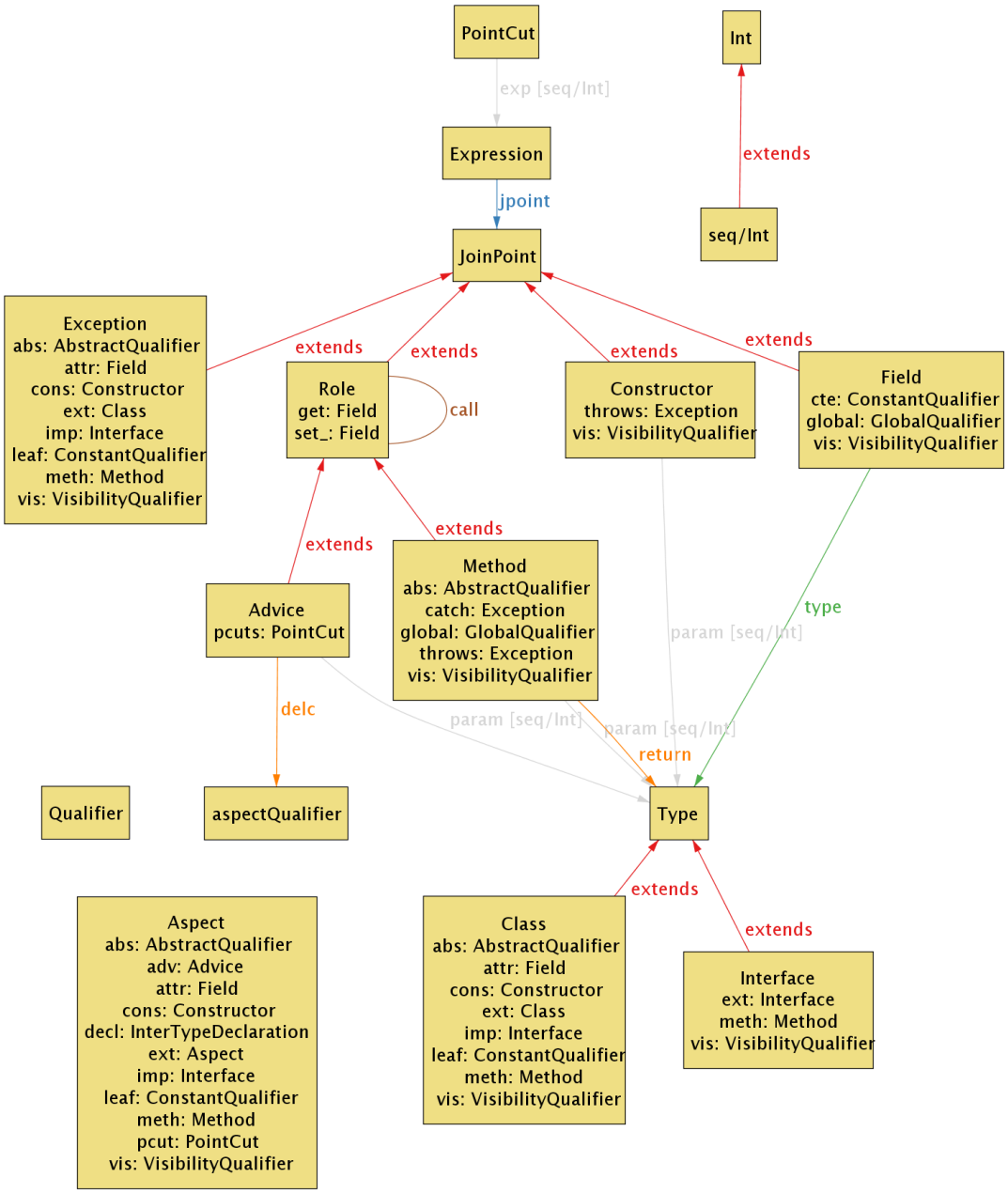


Figure A.1: Class Diagram of the Alloy Theory Module

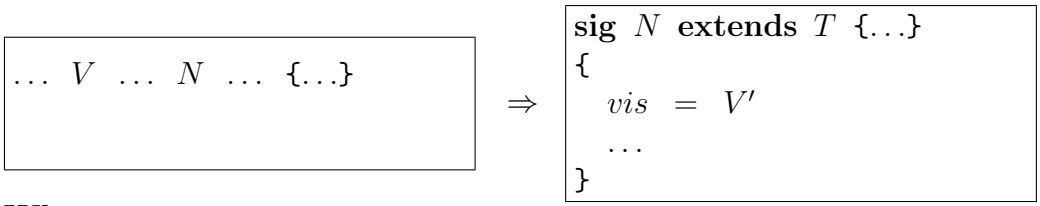
A.2 Translations

For each signature present in our theory module there exists a set of translations that map design rules to an Alloy model based on the theory. In each subsection, we explain one translation or a set of translations, that leads to signature(s), relation(s) and attached fact(s) that corresponds to the design rule semantics. In summary, we translate to an alloy model the design rules written using LSD.

A.2.1 Visibility

Translation 1 shows how class, interface, aspect, method (ITD also), attribute (ITD also), constructor and pointcut visibility is translated to the *vis* relation in the corresponding signatures.

Translation 1 ⟨Visibility⟩



Where:

V = public, protected, private or package (when not specified)

V' = public, protected, private_ or friendly (based on V)

N = name

T = class, interface, aspect, method, constructor, attribute or pointcut

A.2.2 Abstract

Translation 2 shows how the *abstract* qualifier from a class, interface, aspect, method or pointcut is translated to Alloy.

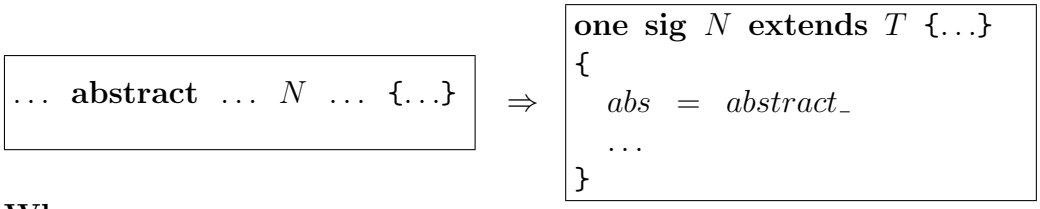
Similarly, whenever we have a negation of the abstract qualifier (*!abstract*), the translation is to *non_abstract*.

A.2.3 Final

Translation 3 shows how the final qualifier from a class, aspect, method or attribute is translated to Alloy.

Similarly, whenever we have a negation of the final qualifier (*!final*), the translation is to *non_final*.

Translation 2 ⟨Abstract⟩

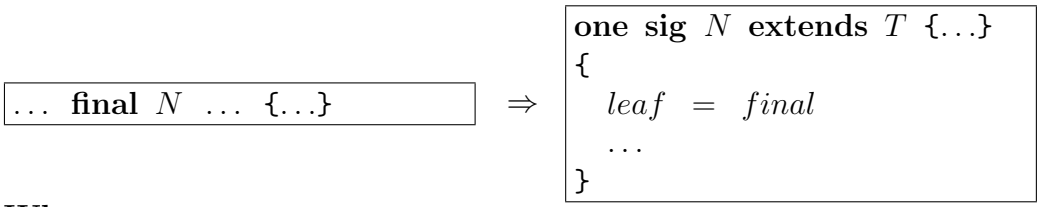


Where:

N = name

T = class, interface, aspect, method or pointcut

Translation 3 ⟨Final⟩



Where:

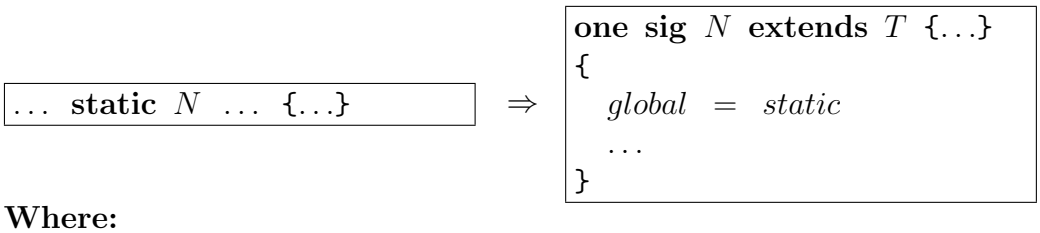
N = name

T = class, aspect, method or attribute (also ITD)

A.2.4 Static

The translation of the static qualifier from a class, interface, aspect, methods or attributes is presented in Translation 4. It is similar to final Translation 3 and abstract Translation 2 translations, including the negation possibility, which is translated to *global = non static* instead of *global = static*.

Translation 4 ⟨Static⟩



Where:

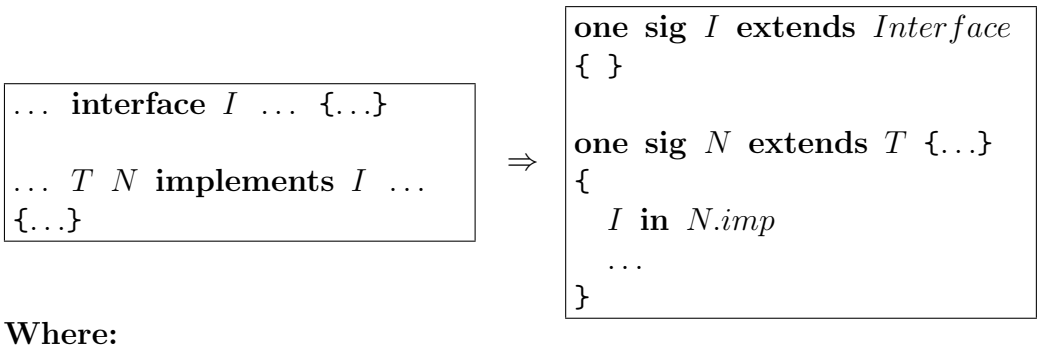
N = name

T = class, aspect, method or attribute (also ITD)

A.2.5 Implements

For interface implementation translation, we need to define relations between different signatures, as shown in Translation 5 through the definition of an interface in Alloy.

Translation 5 ⟨Implements⟩



Where:

N = name

T = class or aspect

If we have a negation of an interface in the implements, the rule indicates that the class or aspect cannot implement the interface. This is expressed in Alloy by changing *in* by *not in*.

A.2.6 Inheritance

Translation 6 shows how to translate inheritance relationship to Alloy. It important to notice that classes and aspect can only have one ancestor, but interfaces can inherit from several other interfaces.

Translation 6 ⟨Inheritance⟩

```
... T N ... {...}

... T N' extends N ...
{...}
```

⇒

```
one sig N extends T {...}
one sig N' extends T {...}
{
  N in ext
  ...
}
```

Where:

N = name

N' = name

T = class or aspect

If we have a negation of an interface in the extends, the rule indicates that the class or aspect cannot extend the class. This is expressed in Alloy by changing *in* by *not in*.

A.2.7 Exceptions

Methods and constructors can raise exceptions. As a result, we create signatures corresponding to the exceptions, in case of they do not exist. Then we add these signatures (exceptions) to the set of exceptions thrown by the method or constructor, as shown in Translation 7.

Translation 7 ⟨Exception⟩

```
... N (..) throws Ex {...}
```

⇒

```
one sig Ex
    extends Throwable {}

one sig N extends T {...}
{
  Ex in throws
}
```

Where:

N = name

T = method, constructor (including ITD)

Similarly, we can exchange *throws* by *catch* in Translation 7 to show how to represent the *set of caught exceptions* in the body of a method or constructor. For both translations we can use the negation operator to indicate that the set of exceptions raised or caught cannot include a certain exception.

A.2.8 Attribute

Translation 8 shows how to translate an attribute from a class, interface or aspect to Alloy. Since attibutes are related to the component in which they are defined, in Alloy we preserved this pattern by creating one signature that corresponds to the attribute (extending Field signature) and a relation to it in the signature that corresponds to the component in which the attribute is declared. This procedure is performed to each attribute.

Translation 8 ⟨Attribute⟩

```
... T N ... {
  T' N';
  ...
}
```

⇒

```
one sig N extends T {...}
one sig N'
      extends Field {...}
{
  N' in N.attr
  type = T'
  ...
}
```

Where:

N = name

N' = attribute name

T = class, interface or aspect

T' = attribute type

If we have a negation of the attribute, we add a rule indicating that the component does not have the attribute, through the use of the *not in* operator instead of *not*.

A.2.9 Pointcut

LSD supports the definition of pointcuts without the expression part. In this case the DR indicates that de aspect developer must follow the constraints included in the DR, like pointcut name and the other qualifiers associated to it, but is free to define the pointcut expression.

Translation 9 express that for each pointcut (without the pointcut expression), it will be created a new signature that extends the *Pointcut* signature and add it to the relation *pcut* from the corresponding *Aspect*, indicating that it is defined within that component.

Translation 9 ⟨Pointcut without Expression⟩

```
... aspect N ... {
  pointcut P();
  ...
}
```

⇒

```
one sig N
      extends Aspect {}

one sig P
      extends Pointcut {...}
{
  P in N.pcut
  ...
}
```

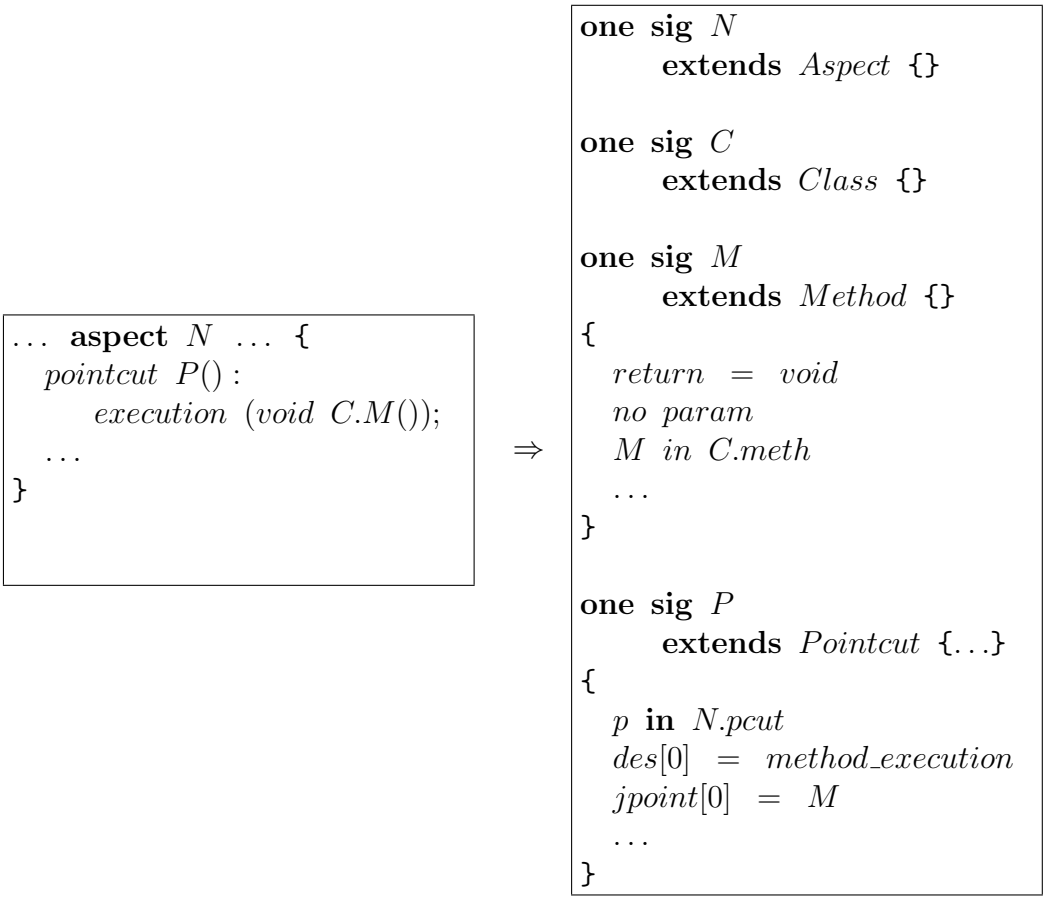
Since a DR can contain aspects with pointcuts containing pointcut expressions, we have to choose a method to determine if a pointcut declared in an aspect satisfies the constraints associated to the DR it implements. This was discussed in Section 3.2.12, and we choose to adopt the *Syntactically Equal* approach but allowing the aspect developers to omit the pointcut declaration and automatically inherit the declaration from the design rule.

A DR can contain aspects that declare pointcuts with expressions. In order to check if an aspect implementing the DR provides a correct pointcut, we require that the expression is Syntactically Equal to the expression declared in the DR. Translation 10 shows that a pointcut is converted to a signature that extends Pointcut signature. It includes a sequence of designators and another to the join point. Based on these sequences, we compare with pointcuts defined in the aspects an check if they match. In this translation it is important to observe that the signatures to method M and class C in which it is declared is created only if it is not already defined.

The pointcut designators available in AspectJ that are supported by our translations to Alloy are:

- method_execution: Method execution (*execution*);
- method_call: Method call (*call*);
- constructor_execution: Constructor execution (*execution*);
- constructor_call: Constructor call (*call*);
- class_init: Class intialization (*initialization*);
- field_read: Attribute access (*get*);
- field_access: Attribute change (*set*);
- exception_handler: Exception handling (*handler*);
- advice_execution: Advice execution (*adviceexecution*).

Translation 10 ⟨Pointcut with Expression⟩

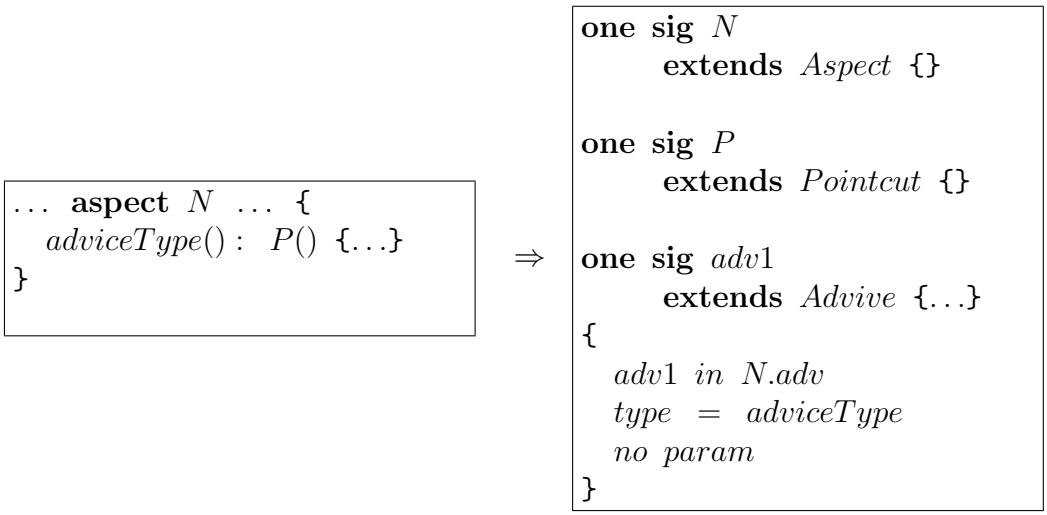


A.2.10 Advice

Advice are associated to a pointcut expression, that indicates when the advice body must be executed. As a result they are not callable and for this reason have no name. LSD supports advice declarations in aspects within DRs. So it is necessary to define a method to guarantee that the implementing aspect defines the expected advice.

Translation 11 shows how constraints associated to advice are expressed in Alloy. Basically, the pointcut must match, like we defined in Section A.2.9, the advice type must be equal and parameters follow the same rules from method parameters. An important thing to notice is that we generate an arbitrary name to the advice so that we can refer from for example facts and other signatures.

Translation 11 ⟨Advice⟩



Appendix B

BNF Specification of LSD

Below we present the complete specification of the *Abstract Syntax Tree* (AST) of LSD. We define it using the JastAdd notation generate the AST nodes, which we describe in Section 5.3.2.

```
Program ::= CompilationUnit*;

CompilationUnit ::= LSDDecl* DRDecl* DRIDecl*;

abstract LSDDecl;

// ------------ Design Rule Instance Declaration -------------

DRIDecl: LSDDecl ::= <ID:String> <DRName:String> [DRIParamIdList];

DRIParamIdList ::= DRIParamId*;

DRIParamId ::= <ID:String> DRIIdSet;

DRIIdSet ::= DRIIdExpr*;

DRIIdExpr ::= IdPattern:DRIIdPattern <Not:boolean>;

// ------------------- Design Rule Declaration ---------------

DRDecl : LSDDecl ::= <ID:String> DRParam* StructRuleDecl*;

DRParam ::= <ID:String>;

// ---------------------- Structural Rules ------------------

StructRuleDecl ::= [ModifierExpr] [TypeExpr] <ID:String>
Extends:IdPatternSet* Implements:IdPatternSet* BodyDecl*;
```

```
// ----------------------- Modifiers ------------------

abstract ModifierExpr;

abstract BinaryModifierExpr : ModifierExpr ::= Lhs:ModifierExpr
Rhs:ModifierExpr;

OrModifierExpr : BinaryModifierExpr; //Modifier || Modifier

AndModifierExpr : BinaryModifierExpr; //Modifier && Modifier

XOrModifierExpr : BinaryModifierExpr; //Modifier ^^ Modifier

NegModifierExpr : ModifierExpr ::= ModifierExpr; //!Modifier

BasicModifierExpr : ModifierExpr ::= Modifier;

Modifier ::= <ID:String>;

// ----------------------- TypeExpr ------------------

abstract TypeExpr;

abstract BinaryTypeExpr : TypeExpr ::= Lhs:TypeExpr Rhs:TypeExpr;

OrTypeExpr : BinaryTypeExpr; //Type || Type

AndTypeExpr : BinaryTypeExpr; //Type && Type

XOrTypeExpr : BinaryTypeExpr; //Type ^^ Type

NegTypeExpr : TypeExpr ::= TypeExpr; //!Type

BasicTypeExpr : TypeExpr ::= <ID:String>;

// ----------------------- Body Declarations ------------------

abstract BodyDecl;

BehavioralRuleDecl : BodyDecl ::= RuleExpr;

QuantificationOperator : BodyDecl ::= Scope: QuantOpMemberDecl*
Rule:QuantOpMemberDecl*;

abstract MemberDecl: BodyDecl ::= [RuleBlock];
```

```
FieldDecl : MemberDecl ::= [ModifierExpr] Type:IdPattern
VariableDecl*;

MethodDecl : MemberDecl ::= [ModifierExpr] Type:IdPattern
ID:IdPattern Parameter:ParameterDecl* Exception:IdPatternSet*;

ConstructorDecl : MemberDecl ::= [ModifierExpr]
Parameter:ParameterDecl* Exception:IdPatternSet*;

abstract IntertypeDecl : MemberDecl;

AttITD         : IntertypeDecl ::= AttITDSig;

MethodITD      : IntertypeDecl ::= MethodITDSig;

ConstructorITD : IntertypeDecl ::= ConstructorITDSig;

abstract DeclareDeclaration : MemberDecl;

VariableDecl ::= ID:IdPattern;

ParameterDecl ::= [ModifierExpr] Type:IdPattern;

RuleBlock ::= BehavioralRuleDecl*;

QuantOpMemberDecl ::= MemberDecl <Op:String>;

// ------------------- Id Pattern Set -------------------

abstract IdPatternSet ::= IdPattern* <EmptySet:boolean>
<Explicit:boolean>;

ExactlyIdPatternSet        : IdPatternSet;

IncludesIdPatternSet       : IdPatternSet;

ExcludesIdPatternSet       : IdPatternSet;

// -------------- Quantification Operators ---------------

ExistsQuantOperator : QuantificationOperator;

AllQuantOperator : QuantificationOperator;

OneQuantOperator : QuantificationOperator;
```

```
NoneQuantOperator : QuantificationOperator;

OptQuantOperator : QuantificationOperator;

RangeQuantOperator : QuantificationOperator ::= <Min:String>
<Max:String>;

// ---------------------- Rules ------------------------

abstract RuleExpr;

abstract BinaryRuleExpr : RuleExpr ::= Lhs:RuleExpr Rhs:RuleExpr;

OrRuleExpr  : BinaryRuleExpr;   //Rule || Rule

AndRuleExpr : BinaryRuleExpr;   //Rule && Rule

XOrRuleExpr : BinaryRuleExpr;   //Rule ^^ Rule

NegRuleExpr : RuleExpr ::= RuleExpr; //!Rule

// ---------------------- MemberExpr ------------------

abstract MemberExpr : BodyDecl;

abstract BinaryMemberExpr : MemberExpr ::= Lhs:MemberExpr
Rhs:MemberExpr;

OrMemberExpr : BinaryMemberExpr;   //MemberExpr || MemberExpr

AndMemberExpr : BinaryMemberExpr;  //MemberExpr && MemberExpr

XOrMemberExpr : BinaryMemberExpr;  //MemberExpr ^^ MemberExpr

NegMemberExpr : MemberExpr ::= MemberExpr;  //!MemberExpr

BasicMemberExpr : MemberExpr ::= BodyDecl;

// -------------------- Behavioral Rules -----------------

CallRuleExpr             : RuleExpr ::= MethodITDSig;

CallConstructorRuleExpr  : RuleExpr ::= ConstructorITDSig;

GetRuleExpr              : RuleExpr ::= AttITDSig;
```

```
SetRuleExpr              : RuleExpr ::= AttITDSig;

XCallRuleExpr            : CallRuleExpr;

XCallConstructorRuleExpr : CallConstructorRuleExpr;

XGetRuleExpr             : GetRuleExpr;

XSetRuleExpr             : SetRuleExpr;

// -------------------- ITD Signature -------------------

abstract ITDSig ::= [ModifierExpr];

AttITDSig : ITDSig ::= Type:IdPattern HostType:IdPattern
ID:IdPattern;

abstract MetConITDSig : ITDSig ::= TargetType:IdPattern

Parameter:ParameterDecl* Exception:IdPatternSet*;

MethodITDSig : MetConITDSig ::= Type:IdPattern ID:IdPattern;

ConstructorITDSig : MetConITDSig;

//------------- Identifier Expression -------------------

abstract IdPattern;

abstract BinaryIdPattern : IdPattern ::= Lhs: IdPattern Rhs:
IdPattern;

AndIdPattern : BinaryIdPattern;

OrIdPattern : BinaryIdPattern;

XOrIdPattern : BinaryIdPattern;

NegIdPattern : IdPattern ::= IdPattern;

ArrayIdPattern : IdPattern ::= IdPattern;

Dims;

abstract IdentifierPattern : IdPattern ::= <IdPattern:String>;
```

```
NameIdPattern : IdentifierPattern;

DRIIdPattern : IdentifierPattern;

DotDotPattern : NameIdPattern;

SubtypeIdPattern : IdPattern ::= IdPattern;

//---------------------- Aspect Members --------------------

PointcutDecl : MemberDecl ::= [ModifierExpr] ID:IdPattern
Parameter:ParameterDeclaration* PointcutExpr;

AdviceDecl   : MemberDecl ::= [ModifierExpr] AdviceSpec
PointcutExpr Exception:IdPatternSet*;

ParameterDeclaration ::= [ModifierExpr] Type:IdPattern
ID:IdPattern;

//------------------------ Advice ------------------------

abstract AdviceSpec ::= Parameter:ParameterDeclaration*;

BeforeSpec : AdviceSpec;

AfterSpec  : AdviceSpec;

AroundSpec : AdviceSpec ::= ReturnType:NameIdPattern;

AfterReturningSpec : AfterSpec ::=
[ReturnParameter:ParameterDeclaration];

AfterThrowingSpec  : AfterSpec ::=
[ExceptionParameter:ParameterDeclaration];

//-------------------- Pointcut Expressions ------------------

abstract PointcutExpr;

abstract BinaryPointcutExpr : PointcutExpr ::= Lhs:PointcutExpr
Rhs:PointcutExpr;

OrPointcutExpr : BinaryPointcutExpr;

AndPointcutExpr : BinaryPointcutExpr;
```

```
NegPointcutExpr : PointcutExpr ::= PointcutExpr;

//-------------------- Pointcut Designators -----------------

AdviceExecutionPointcutExpr : PointcutExpr;

EmptyPointcutExpr : PointcutExpr; //used by abstract pointcuts

CflowPointcutExpr : PointcutExpr ::= PointcutExpr:PointcutExpr;

CflowBelowPointcutExpr : PointcutExpr ::=
PointcutExpr:PointcutExpr;

GetPointcutExpr : PointcutExpr ::= Pattern:AttITDSig;

SetPointcutExpr : PointcutExpr ::= Pattern:AttITDSig;

WithinPointcutExpr  : PointcutExpr ::= Pattern:IdPattern;

HandlerPointcutExpr : PointcutExpr ::= Pattern:IdPattern;

StaticInitializationPointcutExpr : PointcutExpr ::=
Pattern:IdPattern;

ThisPointcutExpr   : PointcutExpr ::= Pattern:IdPattern;

TargetPointcutExpr : PointcutExpr ::= Pattern:IdPattern;

ArgsPointcutExpr   : PointcutExpr ::= Pattern:IdPattern*;

InitializationPointcutExpr : PointcutExpr ::=
Pattern:ConstructorITDSig;

PreInitializationPointcutExpr : PointcutExpr ::=
Pattern:ConstructorITDSig;

CallPointcutExpr       : PointcutExpr ::= Pattern:MetConITDSig;

ExecutionPointcutExpr  : PointcutExpr ::= Pattern:MetConITDSig;

WithinCodePointcutExpr : PointcutExpr ::= Pattern:MetConITDSig;

NamedPointcutExpr : PointcutExpr ::= Name:NameIdPattern
IdPattern*;
```

```
//-------------------- Declare Declarations ----------------------

DeclareParentsExtends : DeclareDeclaration ::= Target:IdPattern
Extends:IdPattern*;

DeclareParentsImplements : DeclareDeclaration ::= Target:IdPattern
Implements:IdPattern*;

DeclareError : DeclareDeclaration ::= Pointcut:PointcutExpr
<Message:String>;

DeclareWarning : DeclareDeclaration ::= Pointcut:PointcutExpr
<Message:String>;

DeclareSoft : DeclareDeclaration ::= Exception:NameIdPattern
Pointcut:PointcutExpr;

DeclarePrecedence : DeclareDeclaration ::= Type:IdPattern*;
```

Bibliography

[1] Shumpei Akai, Shigeru Chiba, and Muga Nishizawa. Region Pointcut for AspectJ. In *Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS'09)*, pages 43–48, Charlottesville, USA, 2009.

[2] Jonathan Aldrich. Open Modules: Modular Reasoning About Advice. In *Proceedings of 19th European Conference on Object-Oriented Programming (ECOOP'05)*, pages 144–168. Springer, 2005.

[3] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall Ptr, 2003.

[4] Vander Alves. *Implementing Software Product Line Adoption Strategies*. PhD thesis, Informatics Center, Federal Universisty of Pernambuco, Recife, Brazil, March 2007.

[5] Vander Alves, Pedro Matos Jr., and Paulo Borba. An Incremental Aspect-Oriented Product Line Method for J2ME Game Development. In *Workshop on Managing Variability Consistently in Design and Code, in conjunction with the 19th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, October 2004.

[6] Vander Alves, Pedro Matos Jr., Leonardo Cole, Paulo Borba, and Geber Ramalho. Extracting and Evolving Mobile Games Product Lines. In *Proceedings of the 9th International Software Product Line Conference (SPLC'05)*, volume 3714 of *Lecture Notes in Computer Science*, pages 70–81. Springer-Verlag, September 2005.

[7] Vander Alves, Alberto Costa Neto, Sérgio Soares, Gustavo Santos, Fernando Calheiros, Vilmar Nepomuceno, Davi Pires, Jorge Leal, and Paulo Borba. From Conditional Compilation to Aspects: A Case Study in Software Product Lines Migration. In *Aspect-Oriented Product Line Engineering AOPLE'06, Workshop of the Fifth International Conference on Generative Programming and Component Engineering (GPCE'06)*, USA, 2006.

[8] Tomoyuki Aotani and Hidehiko Masuhara. SCoPE: an AspectJ Compiler for Supporting User-Defined Analysis-Based Pointcuts. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD'07)*, pages 161–172, New York, NY, USA, 2007. ACM.

[9] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Oege De Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. abc : An extensible AspectJ compiler. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, pages 87–98. ACM Press, 2005.

[10] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondrej Lhoták, Oege de Moor, and Damien Sereni. Building the abc AspectJ compiler with Polyglot and Soot. Technical Report abc-2004-4, Oxford University, December 2004.

[11] Pavel Avgustinov, Torbjörn Ekman, and Julian Tibble. Modularity First: A Case for Mixing AOP and Attribute Grammars. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development (AOSD'08)*, pages 25–35, New York, NY, USA, 2008. ACM.

[12] Carliss Y. Baldwin and Kim B. Clark. *Design Rules: The Power of Modularity Volume 1*. MIT Press, Cambridge, MA, USA, 1999.

[13] Marc Bartsch and Rachel Harrison. A Coupling Framework for AspectJ. In *EASE 2006*, 2006.

[14] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change (2nd Edition)*. Addison-Wesley Professional, 2004.

[15] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[16] João Brunet, Dalton Guerrero, and Jorge Figueiredo. Design Tests: An Approach to Programmatically Check your Code Against Design Rules. In *31st International Conference on Software Engineering (ICSE 2009), New Ideas and Emerging Results*, May 2009.

[17] Mariano Ceccato and Paolo Tonella. Measuring the Effects of Software Aspectization. In *1st Workshop on Aspect Reverse Engineering*, 2004.

[18] Christina Chavez, Alessandro Garcia, Uirá Kulesza, Claudio Sant'Anna, and Carlos José Pereira de Lucena. Crosscutting Interfaces for Aspect-Oriented Modeling. In *Journal of the Brazilian Computer Society*, volume 12, pages 43–58, 2006.

[19] Kung Chen and Chin-Hung Chien. Extending the Field Access Pointcuts of AspectJ to Arrays. In *20th International Conference on Supercomputing (ICS'06)*, pages 470–475, New York, NY, USA, June 2006. ACM Press.

[20] Shyam R. Chidamber and Chris F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[21] Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[22] Curtis Clifton and Gary T. Leavens. Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning. In *Foundations of Aspect Languages (FOAL'2002), in conjunction with the 1st International Conference on Aspect-Oriented Software Development (AOSD'02)*, pages 33–44, 2002.

[23] Roberta Coelho, Vander Alves amd Uirá Kulesza, Alberto Costa Neto, Alessandro Garcia, Arndt von Staa, Carlos Lucena, and Paulo Borba. On Testing Crosscutting Features using Extension Join Points. In *3rd International Workshop on Software Product Line Testing (SPLIT 2006), in conjunction with 10th International Software Product Line Conference (SPLC 2006)*, August 22 2006.

[24] Leonardo Cole and Paulo Borba. Deriving Refactorings for AspectJ. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD'05)*, pages 123–134, New York, NY, USA, 2005. ACM.

[25] Ayla Dantas, Vander Alves, and Paulo Borba. Using Aspects to Sstructure Small Devices Adaptive Applications. In *First Workshop on Reuse in Constrained Environments, in conjunction with the 18th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'03)*, October 2003.

[26] Ayla Dantas and Paulo Borba. Developing Adaptive J2ME Applications Using AspectJ. In *VII Brazilian Symposium on Programming Languages (SBLP'03)*, pages 226–242, May 2003.

[27] Victor Hugo de Carvalho Fernandes. CrossMDA2: Uma abordagem para minimizar o problema da fragilidade de pointcuts na evolução de sistemas orientados a aspectos. Master's thesis, UFRN, July 2009.

[28] Alexander Demenchuk. Beaver. At http://beaver.sourceforge.net/, February 2010.

[29] Cornell University Computer Science Department. Polyglot Extensible Compiler Framework. At http://www.cs.cornell.edu/projects/polyglot/, February 2010.

[30] Premkumar Devanbu, Bob Balzer, Don Batory, Gregor Kiczales, John Launchbury, David Parnas, and Peri Tarr. Modularity in the New Millenium: A Panel Summary. In *Proceedings of the 25th International Conference on Software Engineering (ICSE'03)*, pages 723–724, Washington, DC, USA, 2003. IEEE Computer Society.

[31] Marcos Dósea, Alberto Costa Neto, Paulo Borba, and Sérgio Soares. Specifying Design Rules in Aspect-Oriented Systems. In *First Latin American Workshop on Aspect-Oriented Software Development (LA-WASP'07), in conjuction with the 21st Brazilian Symposium on Software Engineering (SBES'07)*, João Pessoa, Brazil, oct 2007.

[32] Eclipse.org. Eclipse. At http://www.eclipse.org, February 2010.

[33] Eclipse.org. Eclipse Developer Guide: ajc - the AspectJ compiler/weaver. At http://www.eclipse.org/aspectj/doc/next/devguide/ajc-ref.html, February 2010.

[34] Torbjörn Ekman and Görel Hedin. The JastAdd Extensible Java Compiler. In *Proceedings of the 22nd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 1–18, New York, NY, USA, 2007. ACM.

[35] Matthias Felleisen. On the Expressive Power of Programming Languages. In *Science of Computer Programming*, pages 134–151. Springer-Verlag, 1990.

[36] Eduardo Figueiredo, Alessandro Garcia, Cláudio Sant'Anna, Uirá Kulesza, and Carlos Lucena. Assessing Aspect-Oriented Artifacts: Towards a Tool-Supported Quantitative Method. In *9th Workshop on Quantative Approaches in Object-Oriented Software Engineering (QAOOSE'05), in conjunction with 19th European Conference on Object-Oriented Programming (ECOOP'05)*, July 25th 2005.

[37] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[38] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1995.

[39] Alessandro Garcia, Cláudio Sant'Anna, Eduardo Figueiredo, Uirá Kulesza, Carlos Lucena, and Arndt von Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. In *LNCS Transactions on Aspect-Oriented Software Development I*, pages 36–74. Springer, 2006.

[40] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition.* Addison-Wesley Longman, Amsterdam, 3 edition, June 2005.

[41] Phil Greenwood, Thiago Bartolomei, Eduardo Figueiredo, Marcos Dosea, Alessandro Garcia, Nelio Cacho, Cláudio Santanna, Sérgio Soares, Paulo Borba, Uirá Kulesza, and Awais Rashid. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, 2007.

[42] Phil Greenwood, Alessandro Garcia, Awais Rashid, Eduardo Figueiredo, Claudio Sant'Anna, Nelio Cacho, Americo Sampaio, Sérgio Soares, Paulo Borba, Marcos Dosea, Ricardo Ramos, Uira Kulesza, Thiago Bartolomei, Monica Pinto, Lidia Fuentes, Nadia Gamez, Ana Moreira, Joao Araujo, Thais Batista, Ana Medeiros, Francisco Dantas, Lyrene Fernandes, Jan Wloka, Christina Chavez, Robert France, and Isabel Brito. On the Contributions of an End-to-End AOSD Testbed. In *Proceedings of the Early Aspects (EARLYASPECTS'07), in conjunction with the 29th International Conference on Software Engineering (ICSE'07)*, page 8, Washington, DC, USA, 2007. IEEE Computer Society.

[43] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, 23(1):51–60, 2006.

[44] Bruno Harbulot and John R. Gurd. A join point for loops in AspectJ. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, pages 63–74, New York, NY, USA, 2006. ACM.

[45] William Harrison, Charles Barton, and Mukund Raghavachari. Mapping UML designs to Java. In *Proceedings of the 15th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)*, pages 178–187, New York, NY, USA, 2000. ACM.

[46] Görel Hedin. Generating Language Tools with JastAdd. In *Proceedings of the 3rd Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE'09)*, July 2009.

[47] Kevin Hoffman and Patrick Eugster. Bridging Java and AspectJ through Explicit Join Points. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (PPPJ'07)*, pages 63–72, New York, NY, USA, 2007. ACM.

[48] Lattix Inc. Lattix LDM. At http://www.lattix.com, February 2010.

[49] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT Press, 2006.

[50] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the Evolution of Aspect-Oriented Software with Model-Based Pointcuts. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP'06)*, volume 4067/2006 of *Lecture Notes in Computer Science*, pages 501–525. Springer Berlin, September 2006.

[51] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, October 2001.

[52] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97)*, 1997.

[53] Gregor Kiczales and Mira Mezini. Aspect-Oriented Programming and Modular Reasoning. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)*, pages 49–58, New York, NY, USA, 2005. ACM.

[54] Gerwin Klein. JFlex. At http://jflex.de/, February 2010.

[55] Uirá Kulesza. *Uma Abordagem Orientada a Aspectos para o Desenvolvimento de Frameworks*. PhD thesis, PUC/RJ, Rio de Janeiro, Brazil, April 2007.

[56] Uirá Kulesza, Vander Alves, Alessandro Garcia, Alberto Costa Neto, Elder Cirilo, Carlos J. P. Lucena, and Paulo Borba. Mapping Features to Aspects: A Model-Based Generative Approach. In *Early Aspects: Current Challenges and Future*

Directions, volume 4765/2007 of *Lecture Notes in Computer Science*, pages 155–174. Springer Berlin / Heidelberg, 2007.

[57] Uirá Kulesza, Roberta Coelho, Vander Alves, Alberto Costa Neto, Alessandro Garcia, Ardnt von Staa, Carlos Lucena, and Paulo Borba. Implementing Framework Crosscutting Extensions with EJPs and AspectJ. In *Proceedings of the 20th Brazilian Symposium on Software Engineering (SBES 2006)*, October 2006.

[58] David Larochelle, Karl Scheidt, Kevin Sullivan, Yuan Wei, Joel Winstead, and Anthony Wood. Join Point Encapsulation. In *Workshop on Software-engineering Properties of Languages for Aspect Technologies (SPLAT), in conjunction with 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, March 2003.

[59] Gary T. Leavens. The Java Modeling Language (JML). At http://www.jmlspecs.org, February 2010.

[60] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary Design of JML: A Behavioral Interface Specification Language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

[61] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML. Draft paper, September 2006.

[62] Gary T. Leavens and Curtis Clifton. Multiple Concerns in Aspect-Oriented Language Design: A Language Engineering Approach to Balancing Benefits, with Examples. In *Proceedings of the Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT'07), in conjunction with 6th International Conference on Aspect-Oriented Software Development (AOSD'07)*, page 6, New York, NY, USA, 2007. ACM.

[63] Gary. T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph R. Kiniry, and Patrice Chalin. JML Reference Manual. At http://www.jmlspecs.org/jmlrefman/, February 2010.

[64] Barbara Liskov. Data Abstraction and Hierarchy. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'87) - Addendum*, pages 17–34, New York, NY, USA, 1987. ACM.

[65] Cristina Videira Lopes and Sushil Krishna Bajracharya. Assessing Aspect Modularizations Using Design Structure Matrix and Net Option Value. *LNCS Transactions on Aspect-Oriented Software Development I*, pages 1–35, 2006.

[66] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992.

[67] Bertrand Meyer. *Object-Oriented Software Construction (2nd ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[68] Clint Morgan, Kris De Volder, and Eric Wohlstadter. A Static Aspect Language for Checking Design Rules. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD'07)*, pages 63–72, New York, NY, USA, 2007. ACM Press.

[69] Freddy Muñoz, Olivier Barais, and Benoit Baudry. Vigilant usage of Aspects. In *Proceedings of the Workshop on Aspects, Dependencies, and Interactions (ADI 2007), in conjunction with the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, 2007.

[70] Alberto Costa Neto, Vander Alves, and Paulo Borba. Declaring Static Crosscutting Dependencies in AspectJ. In *3rd Brazilian Workshop on Aspect-Oriented Software Development (WASP'06), in conjunction with the 20th Brazilian Symposium on Software Engineering (SBES'06)*, Florianópolis, Brazil, October 2006.

[71] Alberto Costa Neto, Márcio de Medeiros Ribeiro, Marcos Dósea, Rodrigo Bonifácio, Paulo Borba, and Sérgio Soares. Semantic Dependencies and Modularity of Aspect-Oriented Software. In *1st Workshop on Assessment of Contemporary Modularization Techniques (ACoM'07), in conjunction with the 29th International Conference on Software Engineering (ICSE'07)*, May 2007.

[72] Alberto Costa Neto, Arthur Marques, Rohit Gheyi, Paulo Borba, and Fernando Castor Filho. A Design Rule Language for Aspect-Oriented Programming. In *XIII Brazilian Symposium on Programming Languages (SBLP'09)*, pages 131–144, Gramado-RS, Brazil, August 2009.

[73] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An Extensible Compiler Framework for Java. *12th International Conference on Compiler Construction*, 2622:138–152, 2003.

[74] Massachusetts Institute of Technology. Alloy Analyzer. At http://alloy.mit.edu/alloy4/, February 2010.

[75] Neil Ongkingco, Pavel Avgustinov, Julian Tibble, Laurie Hendren, Oege de Moor, and Ganesh Sittampalam. Adding Open Modules to AspectJ. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, March 2006.

[76] Harold Ossher. Confirmed Join Points. In *Workshop on Software Engineering Properties of Languages and Aspect Technologies (SPLAT), in conjunction with the Fifth International Conference on Aspect-Oriented Software Development (AOSD'06)*, March 2006.

[77] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive Pointcuts for Increased Modularity. In *Proceedings of the 19th European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3586 of *Lecture Notes in Computer Science*, pages 214–240. Springer, July 2005.

[78] D. L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[79] Hridesh Rajan and Gary T. Leavens. Quantified, Typed Events for Improved Separation of Concerns. Technical Report 07-14, Department of Computer Science / Iowa State University, Iowa, USA, July 2007.

[80] Henrique Rebêlo. Implementing JML Contracts with AspectJ. Master's thesis, Department of Computing and Systems, University of Pernambuco, Recife, Brazil, May 2008.

[81] Henrique Rebêlo, Ricardo Lima, Márcio Cornélio, Gary T. Leavens, Alexandre Mota, and César Oliveira. Optimizing JML Feature Compilation in Ajmlc Using Aspect-Oriented Refactorings. In *XIII Brazilian Symposium on Programming Languages (SBLP'09)*, pages 117–130, August 2009.

[82] Henrique Rebêlo, Ricardo Massa Ferreira Lima, Márcio Cornélio, and Sérgio Soares. A JML Compiler Based on AspectJ. In *First International Conference on Software Testing Verification and Validation (ICST'08)*, pages 541–544, April 2008.

[83] Henrique Rebêlo, Sérgio Soares, Ricardo Massa Ferreira Lima, Leopoldo Ferreira, and Márcio Cornélio. Implementing Java Modeling Language Contracts with AspectJ. In *Proceedings of the 23rd Symposium on Applied Computing (SAC'08)*, pages 228–233, 2008.

[84] Márcio Ribeiro, Marcos Dósea, Rodrigo Bonifácio, Alberto Costa Neto, Paulo Borba, and Sérgio Soares. Analyzing Class and Crosscutting Modularity with Design Structure Matrixes. In *Proceedings of 21th Brazilian Symposium on Software Engineering (SBES'07)*, pages 167–181, oct 2007.

[85] Frans Sanen, Eddy Truyen, Wouter Joosen, Andrew Jackson, Andronikos Nedos, Siobhán Clarke, Neil Loughran, and Awais Rashid. Classifying and Documenting Aspect Interactions. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD'06)*, March 2006.

[86] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. Using Dependency Models to Manage Complex Software Architecture. *ACM SIGPLAN Notices*, 40(10):167–176, 2005.

[87] Cláudio Sant'Anna, Alessandro Garcia, Christina Chavez, Carlos Lucena, and Arndt von Staa. On the Reuse and Maintenance of Aspect-Oriented Software: An Assessment Framework. In *XVII Brazilian Symposium on Software Engineering (SBES'03)*, October 2003.

[88] Leonardo Humberto Guimarães Silva. Definição de Conjuntos de Junções Robustos usando Aspect-Aware Interfaces e Aspectos Anotadores. Master's thesis, PUC/Minas, June 2008.

[89] Sérgio Soares. *An Aspect-Oriented Implementation Method*. PhD thesis, Informatics Center, Federal Universisty of Pernambuco, Recife, Brazil, October 2004.

[90] Sérgio Soares, Eduardo Laureano, and Paulo Borba. Implementing Distribution and Persistence Aspects with AspectJ. In *Proceedings of the 17th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 174–190, New York, NY, USA, 2002. ACM.

[91] Friedrich Steimann. The Paradoxical Success of Aspect-Oriented Programming. *Proceedings of the 21st International Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06), ACM SIGPLAN Notices*, 41(10):481–497, 2006.

[92] D. V. Steward. The Design Structure System: A Method for Managing the Design of Complex Systems. *IEEE Transactions on Engineering Management*, 28(3):71–74, 1981.

[93] Maximilian Störzer and Christian Koppen. PCDiff: Attacking the Fragile Pointcut Problem, Abstract. In *European Interactive Workshop on Aspects in Software*, Berlin, Germany, September 2004.

[94] Kevin Sullivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nishit Tewari, and Hridesh Rajan. Information Hiding Interfaces for Aspect-Oriented Design. In *Proceedings of the 10th European Software Engineering Conference and the 13th International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*, pages 166–175, New York, NY, USA, 2005. ACM.

[95] Kevin J. Sullivan, William G. Griswold, Yuanfang Cai, and Ben Hallen. The Structure and Value of Modularity in Software Design. In *Proceedings of the 8th European Software Engineering Conference and the 9th International Symposium on Foundations of Software Engineering (ESEC/FSE'01)*, pages 99–108, New York, NY, USA, 2001. ACM.

[96] Oxford University. abc: The AspectBench Compiler for AspectJ. At http://abc.comlab.ox.ac.uk, February 2010.

[97] Dimitri Van Landuyt, Steven Op de beeck, Eddy Truyen, and Wouter Joosen. Domain-driven Discovery of Stable Abstractions for Pointcut Interfaces. In *Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD'09)*, pages 75–86, New York, NY, USA, 2009. ACM.

[98] Mathieu Verbaere, Elnar Hajiyev, and Oege De Moor. Improve Software Quality with SemmleCode: an Eclipse Plugin for Semantic Code Search. In *22nd Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07) - Demo*, pages 880–881, New York, NY, USA, 2007. ACM.

[99] Ricardo Terra Nunes Bueno Villela. Conformação Arquitetural utilizando Restrições de Dependencia entre Módulos. Master's thesis, PUC/Minas, March 2009.

[100] Dean Wampler. Contract4J for Design by Contract in Java: Design Pattern-Like Protocols and Aspect Interfaces. In *Workshop on Aspects, Components and Patterns for Infrastructure Software (ACP4IS), in conjunction with the 5th Interna-*

tional Conference on Aspect-Oriented Software Development (AOSD'06), March 2006.

[101] Dean Wampler. Aspect-Oriented Design Principles: Lessons from Object-Oriented Design. In *Industry Track - 6th International Conference on Aspect-Oriented Software Development (AOSD'07)*, March 2007.

[102] World Wide Web. Java Compiler Compiler (JavaCC). At http://javacc.dev.java.net/, February 2010.

[103] Lingdong Ye and Kris de Volder. PointcutDoctor: IDE Support for Understanding and Diagnosing Pointcut Expressions. In *Demo - Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD'07)*, March 2007.

[104] Lingdong Ye and Kris de Volder. Pointcut Doctor. At http://pointcutdoctor.sourceforge.net/, February 2010.

[105] Aida Atef Zakaria and Honda Hosny. Metrics for Aspect-Oriented Software Design. In *Third International Workshop on Aspect-Oriented Modeling (AOM), in conjunction with 2nd International Conference on Aspect-Oriented Software Development (AOSD'03)*, 2003.

[106] Jianjun Zhao. Towards A Metrics Suite for Aspect-Oriented Software. Technical Report SE-136-5, Information Processing Society of Japan (IPSJ), Japan, March 2002.

[107] Jianjun Zhao. Measuring Coupling in Aspect-Oriented Systems. In *10th International Software Metrics Symposium (METRICS'2004)*, September 2004.

[108] Jianjun Zhao and Martin Rinard. Pipa: A Behavioral Interface Specification Language for AspectJ. In *Proceedings of the 6th International Conference on Fundamental Approaches to Software Engineering (FASE'2003)*, pages 150–165. Springer-Verlag, April 2003.

[109] Jianjun Zhao and Baowen Xu. Measuring Aspect Cohesion. In *Proceedings of Fundamentals Approaches to Software Engineering (FASE'2004)*, number 2984 in Lecture Notes in Computer Science. Springer-Verlag, March 29-31 2004.